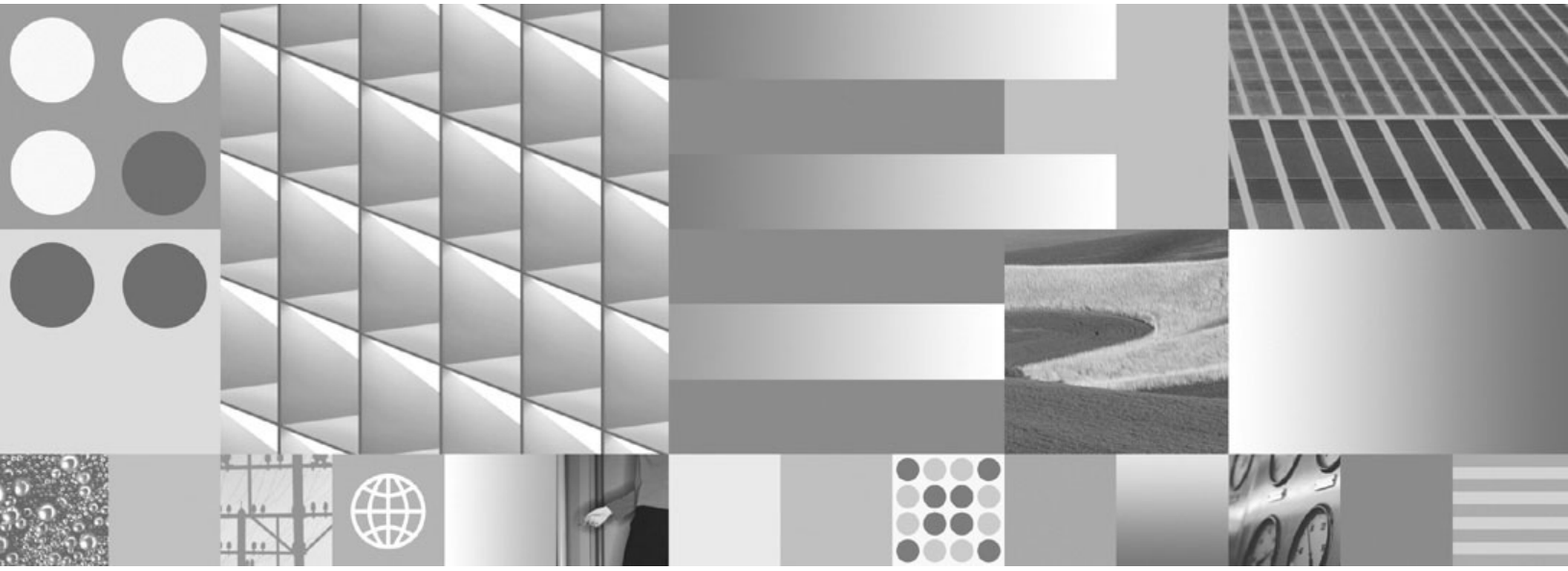
Information Integration

Version 9.1

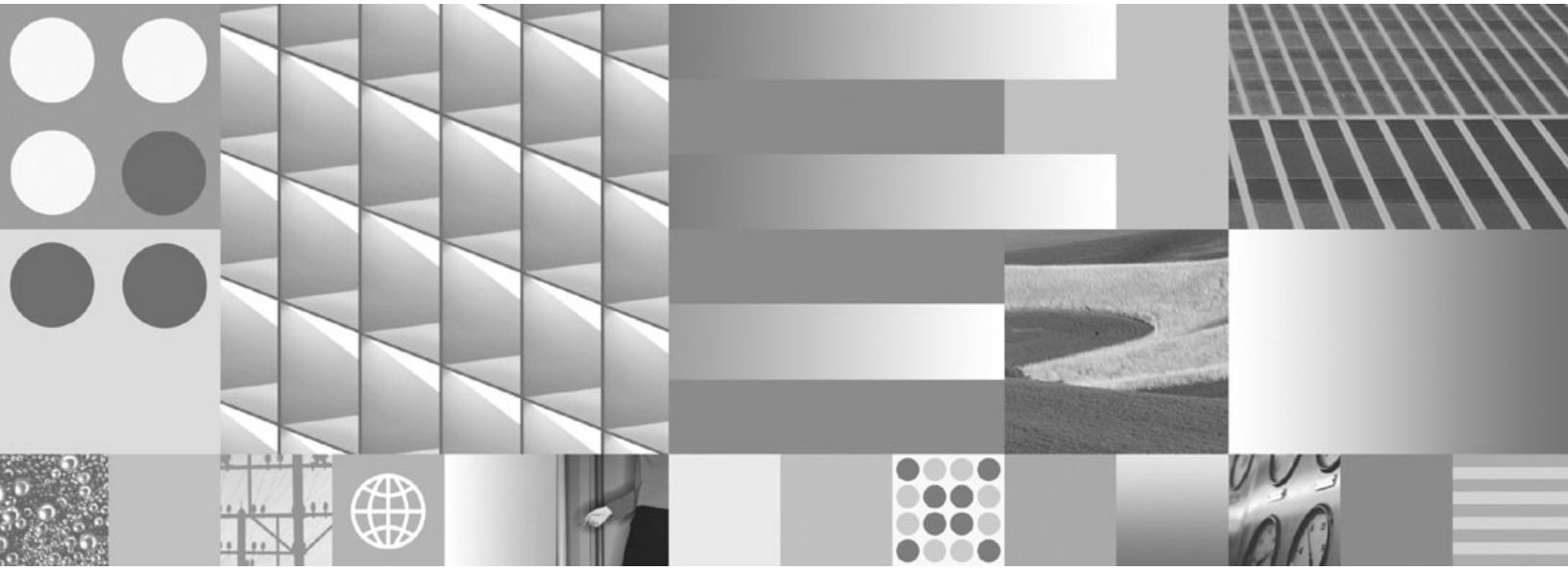


SQL Reference for Classic Federation and Classic Data Event Publishing

SC19-1128-00

Information Integration

Version 9.1



SQL Reference for Classic Federation and Classic Data Event Publishing

SC19-1128-00

Note
Before using this information and the product that it supports, read the information in "Notices" on page 85.

Contents

Chapter 1. General information

You can use the following language elements and syntax diagrams for any of the supported database management systems.

Language elements

The SQL language elements are characters, tokens, SQL statement format, identifiers, naming conventions, authorization IDs, authorization names, qualification of unqualified object names, data types, and constants.

The following topics apply only to the DDL SQL statements for dynamic catalog update operations and to DML SQL statements. See the DB2® SQL reference documentation for the full set of language elements.

Characters

The basic symbols of SQL are characters from the EBCDIC syntactic character set.

These characters are classified as letters, digits, or special characters:

letter Any one of the uppercase alphabetic characters A through Z plus the three EBCDIC code points reserved as alphabetic extenders for national languages (the code points X'5B', X'7B', and X'7C', which display as $, #, and @ in code pages 37 and 500).

digit Any one of the characters 0 through 9.

special character
Any character other than a letter or a digit.

SQL statements can also contain double-byte character set (DBCS) characters. You can use double-byte characters in SQL ordinary identifiers and graphic string constants when you enclose the necessary shift characters. You can also use double-byte characters in string constants and delimited identifiers.

In SQL applications you must contain double-byte characters within a single line. Therefore, you cannot continue a graphic string constant from one line to the next. You can continue a character string constant and a delimited identifier from one line to the next only if the break occurs between single-byte characters.

Tokens

The basic syntactical units of the language are called tokens. A token consists of one or more characters, not including spaces, control characters, or characters within a string constant or delimited identifier.

Tokens are classified as ordinary or delimiter tokens:

Ordinary token
A numeric constant, an ordinary identifier, a host identifier, or a keyword.

```
1
.1
+2
SELECT
E
3
```

Figure 1. Examples of ordinary tokens

Delimiter token
A string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark (?) is also a delimiter token when it serves as a parameter marker.

```
,
'string'
"fld1"
=
.
```

Figure 2. Examples of delimiter tokens

String constants and certain delimited identifiers are the only tokens that can include a space or control character. Any token can be followed by a space or control character. Every ordinary token must be followed by a delimiter token, a space, or a control character. If the syntax does not allow a delimiter token, a space or a control character must follow the ordinary token.

Spaces
A sequence of one or more blank characters. A space is represented as the value x'40'.

Control characters
A special character for string alignment. A control character is treated like a space and does not cause a particular action to occur. The following table identifies the control characters that are supported.

Table 1. Control characters

Name of control character	Hexadecimal value
Tab	05
Form feed	0C
Carriage return	0D
New line or next line	15
Line feed (new line)	25

Uppercase and lowercase
Any token can include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase. Delimiter tokens are never folded to uppercase.

For example, the first statement is equivalent to the second statement, after folding:

```
select * from DSN8710.EMP where lastname = 'Smith';
SELECT * FROM DSN8710.EMP WHERE LASTNAME = 'Smith';
```

SQL statement format

An SQL statement is a complete instruction to the database manager that is written using Structured Query Language.

An SQL statement has a length attribute that identifies the physical length of the SQL statement in bytes. Either the client application explicitly identifies the length of the SQL statement, or the ODBC or JDBC driver determines the length of the SQL statement.

Typically, the SQL statement ends with a null byte, which has a hexadecimal value of zeros. In these cases, the length of the SQL statement is determined by scanning the SQL statement and counting the number of bytes up to the null byte.

SQL identifiers

An identifier is a token that forms a name. An identifier in an SQL statement is an SQL identifier, a parameter marker, or a native identifier. SQL identifiers can be ordinary identifiers or delimited identifiers. They can also be short identifiers, medium identifiers, or long identifiers.

An SQL identifier can be in one of these categories: short ordinary, medium ordinary, long ordinary, short delimited, medium ordinary, or long delimited.

Ordinary identifiers

An ordinary identifier is a letter that is followed by zero or more characters, each of which is a letter, a digit, or the underscore character. An ordinary identifier with an EBCDIC encoding scheme can include Katakana characters.

Double byte character set (DBCS) characters are allowed in SQL ordinary identifiers. You can specify an SQL ordinary identifier, when it is the name of a table, column, view, or stored procedure, by using either DBCS characters or single-byte character set (SBCS) characters. However, an SQL ordinary identifier cannot contain a mixture of SBCS and DBCS characters.

The following rules show how to form DBCS SQL ordinary identifiers. These rules are EBCDIC rules because all SQL statements are in EBCDIC.

- The identifier must start with a shift-out (X'0E') and end with a shift-in (X'0F'). An odd-numbered byte between those shifts must not be a shift-out.
- The maximum length is 8, 18, or 30 bytes including the shift-out and the shift-in depending upon the context of the identifier. In other words, there is a maximum of 28 bytes (14 double-byte characters) between the shift-out and the shift-in.
- There must be an even number of bytes between the shift-out and the shift-in. DBCS blanks (X'4040') are not acceptable between the shift-out and the shift-in.
- The identifiers are not folded to uppercase or changed in any other way.
- Continuation to the next line is not allowed.

An ordinary identifier must not be identical to a keyword that is a reserved word in any context in which the identifier is used.

The following example is an ordinary identifier:

```
SALARY
```

Delimited identifiers

A delimited identifier is a sequence of one or more characters that are enclosed within escape characters. The escape character is the quotation mark (″).

You can use a delimited identifier when the sequence of characters does not qualify as an ordinary identifier. Such a sequence, for example, can be an SQL reserved word, or it can begin with a digit. Two consecutive escape characters represent one escape character within the delimited identifier. A delimited identifier that contains double-byte characters also must contain the necessary shift characters.

When the escape character is the quotation mark, the following example is a delimited identifier:

```
"VIEW"
```

Short, medium, and long identifiers

SQL identifiers are also classified according to their maximum length. A long identifier has a maximum length of 30 bytes. A medium identifier has a maximum length of 18 bytes. A short identifier has a maximum length of 8 bytes. These limits do not include the escape characters of a delimited identifier.

Whether an identifier is long, medium, or short depends on what it represents.

Parameter marker

Parameter markers represent values that are supplied for the SQL statement when it is run. The question mark (?) identifies a parameter marker in an SQL statement.

Native identifiers

Native identifiers exist only in CREATE TABLE and CREATE INDEX statements and refer to a native database object. For example, objects can be an MVS™ data set name, an IMS™ segment name, a CA-IDMS record name, and so on. Native identifiers can be ordinary or delimited identifiers. If the native identifier represents a reserved word, then you must supply a delimited identifier.

The length and allowable characters in a native identifier are DBMS-specific.

Reserved words

A number of words cannot be used as ordinary identifiers in some contexts because these words might be interpreted as SQL keywords.

For example, ALL cannot be a column name in a SELECT statement. Each word, however, can be a delimited identifier in contexts where the word otherwise cannot be an ordinary identifier. For example, the quotation mark (″) is the escape character that begins and ends delimited identifiers. ″ALL″ can appear as a column name in a SELECT statement.

You must use the following words as delimited identifiers where referring to an SQL identifier or native identifier in any SQL statement:

```
ADD
ALL
ALTER
AND
ANY
AS
```

```
AUDIT
BETWEEN
BIND
BUFFERPOOL
BY
CALL
CAPTURE
CHAR
CHARACTER
CHECK
CLUSTER
COLLECTION
COLUMN
CONCAT
CONSTRAINT
COUNT
CURRENT
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURSOR
DATABASE
DAY
DAYS
DEFAULT
DELETE
DESCRIPTOR
DISTINCT
DOUBLE
DROP
EDITPROC
ERASE
ESCAPE
EXCEPT
EXECUTE
EXISTS
FIELDPROC
FOR
FROM
FULL
GO
GOTO
GRANT
GROUP
HAVING
HOUR
HOURS
IMMEDIATE
IN
INDEX
INNER
INOUT
INSERT
INTO
IS
JOIN
KEY
LEFT
LIKE
LOCKMAX
LOCKSIZE
MICROSECOND
MICROSECONDS
MINUTE
MINUTES
MONTH
MONTHS
```

```
NOT
NULL
NUMPARTS
OBID
OF
ON
OPTIMIZE
OR
ORDER
OUT
OUTER
PACKAGE
PART
PLAN
PRECISION
PRIQTY
PRIVILEGES
PROGRAM
REFERENCES
RIGHT
SECOND
SECONDS
SECQTY
SELECT
SET
SOME
STOGROUP
SUBPAGES
SYNONYM
TABLE
TABLESPACE
TO
UNION
UNIQUE
UPDATE
USER
USING
VALIDPROC
VALUES
VCAT
VIEW
VOLUMES
WHERE
WITH
YEAR
YEARS
```

Naming conventions

The rules for forming a name depend on the object type that is designated by the name. The syntax diagrams use different terms for different types of names.

The following list defines terms that represent common SQL objects that are referenced in the various DDL statements.

authorization-name
A short identifier that designates a set of privileges. It can also designate a user or group of users.

column-name
A qualified or unqualified name that designates a column of a table or a view.

A qualified column name is a qualifier followed by a period and a long identifier. The qualifier is a table name or a view name.

An unqualified column name is a long identifier.

index-name
A qualified or unqualified name that designates an index.

A qualified index name is a short identifier followed by a period and a medium identifier. The short identifier is the authorization ID that owns the index.

An unqualified index name is a medium identifier with an implicit qualifier. The implicit qualifier is an authorization ID.

procedure-name
A qualified or unqualified name that designates a stored procedure.

A fully qualified procedure name is a two-part name. The first part is the authorization ID that designates the owner of the procedure. The second part is a medium identifier. A period must separate each of the parts.

A one-part or unqualified procedure name is a medium identifier with an implicit qualifier. The implicit qualifier is an authorization ID.

schema-name
An SQL identifier that designates a schema. A schema name that is used as a qualifier of that object name is often also an authorization ID. The objects that are qualified with a schema name are stored procedures and tables.

table-name
A qualified or unqualified name that designates a table.

A fully qualified table name is a two-part name. The first part is the authorization ID that designates the owner of the table. The second part is a medium identifier. A period must separate each of the parts.

A one-part or unqualified table name is a medium identifier with an implicit qualifier. The implicit qualifier is an authorization ID.

view-name
A qualified or unqualified name that designates a view.

A fully qualified view name is a two-part name. The first part is the authorization ID that designates the owner of the view. The second part is a medium identifier. A period must separate each of the parts.

A one-part or unqualified view name is a medium identifier with an implicit qualifier. The implicit qualifier is an authorization ID.

Authorization IDs and authorization names

An authorization ID is a character string that designates a defined set of privileges. Client connections can successfully run SQL statements only if client connections have the authority to perform the specified functions. A client connection derives this authority from its authorization IDs. An authorization ID can also designate a user or a group of users, but federation does not control this property.

Authorization IDs provide this functionality:

- Authorization checking of SQL statements
- Implicit qualifiers for database objects like tables, views, and indexes

Connections and authorization IDs

Whenever a connection is established, an authorization ID is optionally passed to the server. If an authorization ID is not supplied, the connection is assigned the default authorization ID of PUBLIC. If external security is active by using the SAF EXIT configuration parameter, then as part of connection authentication, the authorization ID is passed to the SAF exit. If the authorization ID is allowed to access the system, the SAF exit can return a secondary authorization ID that is in the list of authorization IDs that are associated with the connection.

Every connection has exactly one primary authorization ID. All other IDs are secondary authorization IDs. A connection can have a maximum of three associated authorization IDs: the authorization ID on the connection request, possibly one secondary authorization ID that is returned by the SAF exit, and the generic authorization ID of PUBLIC.

An authorization name in an SQL statement is not the same as an authorization ID of a connection.

Example

For example, assume that SMITH is the user ID that is supplied during the connection, and you run the following statements:

```
CREATE TABLE TDEPT ...;
GRANT SELECT ON TDEPT TO KEENE;
```

When the GRANT statement is prepared and run, the SQL authorization ID is SMITH. KEENE is an authorization name that is specified in the GRANT statement.

Authorization to run the GRANT statement is checked against SMITH, and SMITH is the implicit qualifier of TDEPT. The authorization rule is that the privilege set is designated by SMITH must include the SELECT privilege with the GRANT option on SMITH.TDEPT. There is no check involving KEENE.

If SMITH is the implicit qualifier for a statement that contains NAME1, then NAME1 identifies the same object as SMITH.NAME1. If the implicit qualifier is other than SMITH, then NAME1 and SMITH.NAME1 identify different objects.

Privileges

For statements other than an ALTER, CREATE, DROP, GRANT, or REVOKE statement, each privilege that is required for the statement can be a privilege that is designated by any authorization ID of the connection. Therefore, a privilege set is the union of the set of privileges that are held by each authorization ID.

If the SQL statement is an ALTER, CREATE, DROP, GRANT, or REVOKE statement, the only authorization ID for authorization checking is the SQL authorization ID. Therefore, the privilege set is the privileges that are held by the single authorization ID that corresponds to the user ID that is supplied on the connection request.

Qualification of unqualified object names

Unqualified object names are implicitly qualified. The rules for qualifying a name differ depending on the type of object that the name identifies.

The implicit qualifier for an unqualified index, table, or view name is the authorization ID in the CURRENT SQLID special register. This authorization ID corresponds to the user ID of the connected user that runs the SQL statement.

Data types

The smallest unit of data that can be manipulated in SQL is called a value. How values are interpreted depends on the data type of their source.

The sources of values are as follows:

- Columns
- Constants
- Expressions
- Host variables
- Special registers

All data types include the null value. Distinct from all non-null values, the null value is a special value that denotes the absence of a (non-null) value. Although all data types include the null value, some sources of values cannot provide the null value. For example, constants, columns that are defined as NOT NULL, and special registers cannot contain null values. The COUNT function cannot return a null value column as the result of a query.

Character string

A character string is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is an empty string. An empty string is not the same thing as a null value.

The bytes of a character string can represent a mixture of characters from a single-byte character set (SBCS) and a double-byte character set (DBCS). Strings that might contain both SBCS and DBCS characters are called mixed data. EBCDIC mixed data might contain shift characters, which do not represent SBCS or DBCS data.

The following subtypes of character strings are supported:

Fixed-length character strings

All the values of a column with a fixed-length character-string data type have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 255. Every fixed-length string column is a short string column. A fixed-length character-string column can also be called a CHAR or CHARACTER column.

Varying-length character strings

All values of varying-length string columns have the same maximum length, which is determined by the length attribute. If the length attribute is greater than 254, the column is a long-string column. Long-string columns cannot be referenced in these items:

- Function other than SUBSTR or LENGTH
- GROUP BY clause
- ORDER BY clause
- CREATE INDEX statement
- SELECT DISTINCT statement
- Subselect of a UNION without the ALL keyword

- Predicate other than LIKE

The VARCHAR column identifies a short, varying-length string column. A maximum-length attribute must be specified, and must be between 1 and 32704. The VARCHAR(*x*) syntax is preferred over LONG VARCHAR, which is a long-string column that does not have an explicit length attribute. The maximum length is also 32704. However, a smaller maximum length is explicitly specified or internally computed, based on the maximum physical size limits.

Graphic strings

A graphic string is a sequence of DBCS characters. The length of the string is the number of characters in the sequence. Each character is assumed to be two-bytes long. Like character strings, graphic strings can be empty. An empty string is not the same thing as a null value.

The following subtypes of character strings are supported:

Fixed-length graphic strings
All the values of a column with a fixed-length graphic-string data type have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 127. Every fixed-length graphic-string column is a short string column. A fixed-length, graphic-string column can also be called a GRAPHIC column.

Varying-length graphic strings
All values of varying-length string columns have the same maximum length, which is determined by the length attribute. If the length attribute is greater than 127, the column is a long-string column, and the same rules for LONG VARCHAR columns apply.

The VARGRAPHIC column identifies a short, varying-length graphic-string column, and a maximum-length attribute must be specified. The length attribute must be between 1 and 127. A long-graphic-string column is identified as a LONG VARGRAPHIC column and does not have an explicit length attribute. The maximum length is 16352; however, a smaller maximum length is explicitly specified or internally computed, based on the maximum physical-size limits.

Numbers

The numeric data types are binary integer, floating-point, and decimal. Binary integer includes small integer and large integer. Floating-point includes single precision and double precision. Binary numbers are exact representations of integers; decimal numbers are exact representations of real numbers; and floating-point numbers are approximations of real numbers.

All numbers have a sign and a precision. When the value of a column or the result of an expression is a decimal or floating-point zero, its sign is positive. The precision of binary integers and decimal numbers is the total number of binary or decimal digits excluding the sign. The precision of floating-point numbers is either single or double, based on the number of hexadecimal digits in the fraction.

The types of numbers are as follows:

Small integer (SMALLINT)
A small integer is a binary integer with a precision of 15 bits. The range of small integers is -32768 to +32767.

Large integer (INTEGER)

A large integer is a binary integer with a precision of 31 bits. The range of large integers is -2147483648 to +2147483647.

Single precision floating-point (REAL)

A single precision floating-point number is a short (32 bits) floating-point number. The range of single precision floating-point numbers is about -7.2E+75 to 7.2E+75. In this range, the largest negative value is about -5.4E-79, and the smallest positive value is about 5.4E-079.

The query processor uses standard 370 representation to process single precision floating point numbers. Individual databases might treat these numbers in different representations and support different limits. Likewise, when single precision floating point numbers are manipulated by the CLI, JDBC, and ODBC client components, the representation and limits are based on the platform and compiler that is used.

On the z/OS® server, the 370 representation consists of a sign bit, a 7-bit biased hexadecimal exponent, and a 24-bit fractional part. The exponent bias is 64. All operations on single precision floating point numbers are normalized. The value that can be represented by a single precision floating point number is approximately 6 or 7 decimal digits of precision.

Double precision floating-point (DOUBLE or FLOAT)

A double precision floating-point number is a long (64 bits) floating-point number. The range of double precision floating-point numbers is about -7.2E+75 to 7.2E+75. In this range, the largest negative value is about -5.4E-79, and the smallest positive value is about 5.4E-079.

Like single precision numbers, double precision numbers that are manipulated by the z/OS server components use standard 370 representation with the caveat that the source database and client implementation might be different than that used by the data server.

On the z/OS server, the 370 representation consists of a sign bit, a 7-bit biased hexadecimal exponent, and a 56-bit fractional part. The exponent bias is 64. All operations on double precision floating point numbers are normalized. The value that can be represented by a single precision floating point number is approximately 16 or 17 decimal digits of precision.

Decimal (DECIMAL)

A decimal number is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is *-n* to *+n*, where *n* is the largest positive number that can be represented with the applicable precision and scale. The maximum range is $1 - 10^{31}$ to $10^{31} - 1$.

String representations of numbers

Values whose data types are small integer, large integer, floating-point, and decimal are stored in an internal form that is transparent to the user of SQL. But string representations of numbers can be used in some contexts. A valid string representation of a number must conform to the rules for numeric constants.

Constants

A constant (also called a literal) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal. String constants are classified as character or graphic. All constants have the attribute NOT NULL. A negative sign in a numeric constant with a value of zero is ignored.

The types of constants are as follows:

Integer constants

Specifies a binary integer as a signed or unsigned number that has a maximum of 10 significant digits and no decimal point. If the value is not within the range of a large integer, the constant is interpreted as a decimal constant. The data type of an integer constant is large integer.

In syntax diagrams, the term *integer* is used for an integer constant that must not include a sign.

Floating-point constants

Specifies a floating-point number as two numbers separated by an E. The first number can include a sign and a decimal point. The second number can include a sign but not a decimal point. The value of the constant is the product of the first number and the power of 10 that is specified by the second number. The value must be within the range of floating-point numbers. The number of characters in the constant must not exceed 30. Excluding leading zeros, the number of digits in the first number must not exceed 17, and the number of digits in the second must not exceed 2. The data type of a floating-point constant is double precision floating-point.

```
15E1     2.E5    -2.2E-1     +5.E+2
```

Figure 3. Floating-point constants that represent the numbers 150, 200000, -0.22, and 500

Decimal constants

Specifies a decimal number as a signed or unsigned number of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. The precision is the total number of digits, including any to the right of the decimal point. The total includes all leading and trailing zeros. The scale is the number of digits to the right of the decimal point, including trailing zeros.

```
025.50   1000.   -15.   +3758933333333333333.33
```

Figure 4. Decimal constants that have precisions and scales of 5 and 2, 4 and 0, 2 and 0, 23 and 2

Character string constants

Specifies a varying-length character string. Character string constants have these forms:

A sequence of characters that starts and ends with an apostrophe (').

Specifies the character string that is contained between the string delimiters. The number of bytes between the delimiters must not be greater than 255. Two consecutive string delimiters are used to represent one string delimiter within the character string.

An X followed by a sequence of characters that starts and ends with a string delimiter.

Also called a hexadecimal constant. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 254. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. A hexadecimal constant allows you to specify characters that do not have a keyboard representation.

```
'12/14/1985'    '32'    'DON''T CHANGE'    X'FFFF'    ''
```

Figure 5. Examples of character string constants

The last string in the example (’’) represents an empty character string constant, which is a string of zero length.

A character string constant is classified as mixed data if it includes a DBCS substring. In all other cases, a character string constant is classified as SBCS data.

Graphic string constants

Specifies a varying-length graphic string.

In EBCDIC environments, the forms of graphic string constants are:

- G’0x0e[dbcs-string]0x0f’
- N’0x0e[dbcs-string]0x0f’

In SQL statements, graphic string constants cannot be continued from one line to the next. The maximum number of DBCS characters in a graphic string constant is 124.

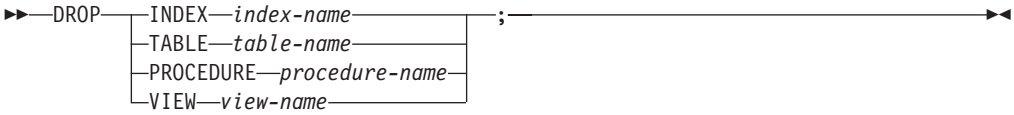
General syntax diagrams

The syntax for the DROP and COMMENT ON statements is the same for all the DBMS types.

DROP statement

The DROP statement deletes an object from the metadata catalog.

Syntax



Parameters

INDEX*index-name*

Deletes an index definition from the metadata catalog.

Following the INDEX keyword is the index name. If qualified, the name is a two-part name, and the authorization ID that qualifies the name is the index

owner. If an unqualified name is supplied, the owner name is the authorization ID from the CURRENT SQLID special register.

One of the following permissions is required to run the statement:

- SYSADM
- DBADM for the database type that is in the DBNAME column for the table that is referenced by the index
- Ownership of *index_name* or the owner of table that is referenced by the index.

TABLE *table-name*

Deletes a table from the metadata catalog. The table name can be a qualified or unqualified table name. If an unqualified name is supplied, the table owner is from the CURRENT SQLID special register.

When a table is deleted, all dependent indexes are deleted, in addition to any views that reference the table. All authorization information that is associated with the table is also deleted from the metadata catalog.

One of the following permissions is required to run the DROP TABLE statement:

- SYSADM
- DBADM for the database type that is in the DBNAME column for the table referenced by the index
- Ownership of the table that you are dropping

PROCEDURE *procedure-name*

Deletes a stored procedure definition from the metadata catalog. The procedure name can be a qualified or unqualified name. If an unqualified name is supplied, the stored procedure owner is from the CURRENT SQLID special register.

All authorization information that is associated with the procedure is also deleted from the metadata catalog.

One of the following permissions is required to run the DROP PROCEDURE statement:

- Ownership of the procedure
- SYSADM authority

VIEW*view-name*

Identifies the name of the view to be deleted. *view-name* can be a qualified or unqualified view name. If an unqualified name is supplied, the view owner is obtained from the CURRENT SQLID special register.

When a view is deleted, all authorization information that is associated with the view is also deleted from the metadata catalog. DROP VIEW deletes dependent views along with those specified in the DROP VIEW statement.

One of the following permissions is required to execute the DROP VIEW statement:

- SYSADM
- Ownership of the view being dropped

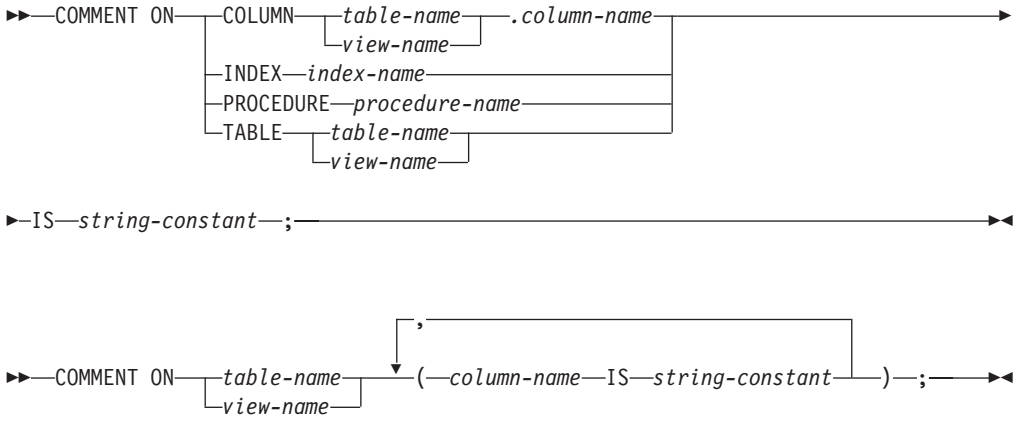
COMMENT ON statement

The COMMENT statement adds or replaces comments in the descriptions of objects in the metadata catalog.

The COMMENT ON statement updates the REMARKS column in the SYSIBM.SYSTABLES, SYSIBM.SYSCOLUMNS, SYSIBM.SYSINDEXES, or SYSIBM.SYSROUTINES table, depending on the form of the statement.

The COMMENT ON statement has two syntax diagrams. The first syntax diagram updates the REMARKS column in a single metadata catalog table. The second syntax diagram updates the REMARKS column in the SYSIBM.SYSCOLUMNS table for a table or view definition.

Syntax



Parameters

COLUMN

Specifies the column that the comment applies to.

The name must identify a column of a table or view that exists in the system catalog. The column names must not be qualified. The comment is placed into the REMARKS column of the SYSIBM.SYSCOLUMNS system table, for the row that describes the column.

Do not use the keywords TABLE or COLUMN to comment on more than one column in a table or view. Give the table or view name and then, in parentheses, a list in the form:

```
column-name IS string-constant,
column-name IS string-constant,...
```

One of the following permissions is required to run the COMMENT ON statement to update a column:

- Ownership of the table or view
- SYSADM authority
- DBADM authority for the database class (only when a table is referenced)

table-name.column-name
Name of the table column that the comment applies to.

view-name.column-name
Name of the view column that the comment applies to.

INDEX *index-name*

Updates the REMARKS column in SYSIBM.SYSINDEXES for an index definition in the metadata catalog.

Following the INDEX keyword is the index name. If qualified, index-name is a two-part name, and the authorization ID that qualifies the name is the owner of the index. If an unqualified index name is supplied, the owner name is the authorization ID, which is from the CURRENT SQLID special register.

One of the following permissions is required:

- Ownership of the table or index
- DBADM authority for the database class of the table referenced by the index
- SYSADM authority

PROCEDURE *procedure-name*

Updates the REMARKS column in SYSIBM.SYSROUTINES table for a stored procedure definition.

Following the PROCEDURE keyword is the procedure name. If qualified, the procedure name is a two-part name, and the authorization ID that qualifies the name is the owner of the procedure. If an unqualified procedure name is supplied, the owner name is the authorization ID from the CURRENT SQLID special register.

One of the following permissions is required:

- Ownership of the procedure
- SYSADM authority

TABLE

Updates the REMARKS column in SYSIBM.SYSTABLES for a table or view.

Following the TABLE keyword is a name that refers to either a table or view. If qualified, the table or view name is a two-part name, and the authorization ID that qualifies the name of the owner of the table or view. If an unqualified table name is supplied, the owner name is the authorization ID that is from the CURRENT SQLID special register.

One of the following permissions is required:

- Ownership of the table or view
- SYSADM authority
- DBADM authority for the database class (only when a table is referenced)

table-name
Name of the table that the comment applies to.

view-name
Name of the view that the comment applies to.

IS *string-constant*

Introduces the comment that you want to make.

string-constant can be any SQL character string constant of up to 254 bytes.

Chapter 2. IMS

You can use the CREATE TABLE, ALTER TABLE and CREATE INDEX statements to define tables and indexes that reference IMS databases.

CREATE TABLE statement for IMS

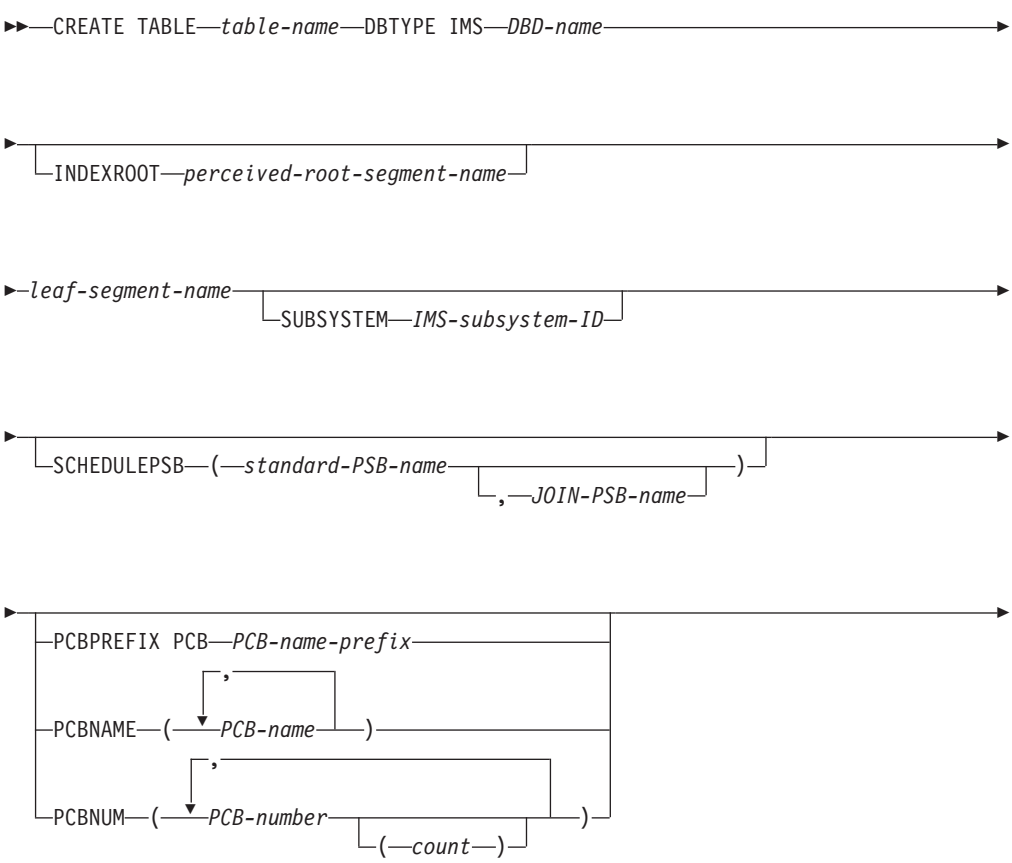
You can use the CREATE TABLE statement to define a logical table that references an IMS database.

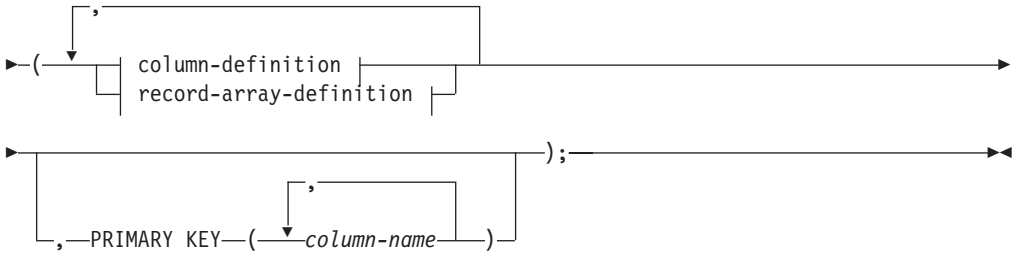
Authorization

The connected user ID must have one of the following privileges to run the CREATE TABLE statement:

- SYSADM
- DBADM for the database type that is referenced in the DBTYPE clause

The owner has all table privileges on the table (such as SELECT, UPDATE, and so on) and the authority to drop the table. The owner can grant equivalent use privileges on the table.





authorization-ID.table-name

Identifies the owner of the table and the name of the table that you want to create.

If you do not provide an authorization ID, the ID in the CURRENT SQLID special register is used.

You must create more than one table if you map to a database or a file that meets either of these criteria:

- The database or file contains repeating data.
- The database or file contains information about one or more distinct sub-objects, because the database or file is not designed to follow the third normalization rules, which are part of the standards to eliminate redundancies and inconsistencies in table data. In a table designed according to third normalization rules, each non-key column is independent of other non-key columns, and is dependent only upon the key.

DBTYPE IMS

Specifies that the CREATE TABLE statement defines a logical table that references an IMS database.

DBD-name

Identifies the IMS DBD (database definition) that the table references. *DBD-name* corresponds to the name in the NAME parameter for the DBD statement. That statement is in the DBDGEN source definition for the IMS logical or physical database that the IMS table references. *DBD-name* follows z/OS load-module naming conventions.

INDEX ROOT *perceived-root-segment-name*

Identifies the root segment for the database hierarchy that the IMS logical table definition maps. By default, the root segment is the physical or logical root segment for the database that *DBD-name* specifies. The physical or logical root segment for the database constitutes the root segment for verification of the IMS hierarchy to the segment that *leaf-segment-name* specifies.

The INDEXROOT clause is required when the logical table references an IMS secondary data structure hierarchy because the intent is to use a secondary index to access or update the IMS database. The INDEXROOT clause is required only when a secondary index is used and the target segment of the secondary index is not the root segment of the database. For additional information about secondary indexes, see the *IMS Administration Guide: Database Manager*.

The name is a short native identifier that follows IMS segment naming conventions. The segment must exist in the DBD that *DBD-name* specifies.

The segment must be either the root segment of the database or the root segment because a secondary index accesses the table and database.

The following caveats apply for the INDEXROOT clause:

- If you define a table for change capture, you cannot define a table that maps to a secondary index and secondary data structure. Changes are captured at the physical DBD level. You cannot use data capture where the INDEXROOT segment is not the root segment of a physical database.
- When a secondary index exists in the database that your table references and the target segment in the segment hierarchy is the root segment of the database, create separate tables for each access path.

  You can use one table to access the IMS database by using the primary key sequence field. You can use the additional tables to access the database by using the secondary index XDFLD definition. Each of these additional tables must use either a different PSB to access the database, or the PCB prefix option if for a single PSB.
- To use a single mapping for both primary key and secondary index access, you must specify the PCB prefix option and explicitly define indexes by using the CREATE INDEX statement for each access technique (primary key or XDFLD). On each index definition, you identify the PCB prefix that selects the PCB that accesses the IMS database.

  In this situation, the PCB that accesses the database depends upon the contents of the WHERE clause:
  - If columns in the WHERE clause provide references to all of the columns that make up an XDFLD definition, the PCB prefix for the index that contains the XDFLD columns is used to access the database.
  - If the WHERE clause contains references to the columns that map to the primary key sequence field, the index that contains the primary key sequence field columns is selected. The PCB prefix that is associated with that index is used to access the database.
  - If the WHERE clause does not contain sufficient information to select either index, the PCB prefix at the table level determines which PCB accesses the database.

leaf-segment-name

Identifies the lowest level segment in the database hierarchy that the table maps to.

The database hierarchy is determined by traversing the parent chain (PARENT= keyword in the DBD definition) for *leaf-segment-name* to either the explicitly identified *perceived-root-segment-name* or the root segment of the physical or logical database.

The name is a short native identifier that follows IMS segment naming conventions. The segment must exist in the DBD that is identified by *DBD-name*.

When *perceived-root-segment-name* is the physical or logical root segment of the database, *leaf-segment-name* must be a physical or logical child of *perceived-root-segment-name*.

When *perceived-root-segment-name* is the root segment in a secondary data structure, *leaf-segment-name* must be a child of *perceived-root-segment-name*.

SUBSYSTEM *IMS-subsystem-ID*

Identifies the IMS subsystem that is the location of the database that *DBD-name* specifies. The Open Database Access (ODBA) interface uses *IMS-subsystem-ID* when it accesses or updates the IMS database for two-phase commit.

The ID is a native identifier that follows IMS subsystem naming conventions. *IMS-subsystem-ID* is 1 to 4 characters in length.

Do not use *IMS-subsystem-ID* for change capture or when an interface other than ODBA accesses the IMS data. If ODBA accesses data and *IMS-subsystem-ID* is not provided, *IMS-subsystem-ID* is obtained from the service information entry for CACRRSI in the configuration file of the data server.

SCHEDULEPSB

Identifies the names of one or two PSBs that access or update the IMS database when the DRA or ODBA interface is used.

Do not use PSB scheduling information in change capture.

standard-PSB-name

Identifies the name of the default PSB that accesses the IMS database.

JOIN-PSB-name

Optionally specifies the name of the PSB that is scheduled to access the IMS database. The DRA and ODBA interfaces use *DBD-name* to identify the IMS database. *JOIN-PSB-name* corresponds to a PSB definition that is defined to the IMS system being accessed and as a PDS member under the same name that exists in the active ACB library of the target IMS subsystem.

If your applications issue joins against multiple IMS tables, specify *JOIN-PSB-name*.*JOIN-PSB-name* follows the naming conventions for the z/OS load module.

The PSB is scheduled when a SELECT statement is run that contains a JOIN predicate that references multiple IMS tables. This first table is the one that JOIN references.

PCBPREFIX PCB *PCB-name-prefix*

Identifies a partial PCBNAME that specifies the PCB that accesses or updates the IMS database.

Do not use PCB prefix for change capture.

The prefix is a native identifier that follows IMS PCB naming conventions. *PCB-name prefix* is from 1 to 7 characters in length.

PCBNAME (*PCB_name*,...)

Specifies up to 5 PCB names, each of which access an IMS database for a table. Multiple names are required if the same table is referenced more than one time in an SQL statement, or when the same PCB name is associated with more than one table, and the additional tables are referenced in a single SQL statement.

Each PCB name in the list is 1 to 8 characters in length.

PCBNUM (*PCB_number* (*count*),...)

Allows more potential PCBs to access the IMS database for the table than the PCBNAME keyword allows. Multiple numbers are required if the same table is referenced more than once in an SQL statement, or when the same PSB is associated with more than one table, the PCBs in the PSB have sensitivity to

the segments that the table accesses and the same PCB ordinal numbers are specified for these tables. These additional tables are referenced in a single SQL statement.

You can specify up to ten sets of PCB number ranges. The order of the numbers represent which order a PSB is checked to determine whether a PCB accesses the IMS database.

For each item in the list, different methods identify the PCB numbers that are checked. The simplest method is separate each PCB number with commas. The second method identifies a range that consists of a starting PCB number that is followed by parenthesis and a number that identifies the number of PCBs to check from the starting number.

With either method, the PCB number represents the relative 1 ordinal number of the PCB that is checked. Because an input or output PCB must be defined in each PSB, the minimum practical PCB number is 2.

column_definition

Provides SQL descriptions of the contents of the segments in the IMS database hierarchy that this table accesses. Optionally, the column can include one or more record arrays that identify repeating data in the sequential file. IMS databases have limited metadata. The DBD definition defines the segments that make up the database and its hierarchical structure.

A table must contain at least one column and can contain up to 5,000 columns. Columns are named, and each name must be unique within the table.

At a minimum, the DBD definition contains an IMS FIELD definition for the keys for each segment in the database and XDFLD definitions that identifies the keys of any secondary indexes. Whether the DBD contains additional FIELD statements that define the other fields in each segment, is site dependent. A common practice is to only define additional FIELD statements for those attributes that are referenced in segment search arguments (SSAs) that the applications issue. Then IMS filters the data that is returned to the application.

record_array_definition

Identifies repeating data. Record array definitions contain column definitions and possibly more record array definitions.

Federated queries can use *record array definitions* if you create a separate table for each array. Change capture can use *record arrays* if you map the columns in a flattened structure that provides a separate column for each array instance and field.

Federated queries

For federated queries, define multiple tables. Each table consists of a single record array definition that contains the column definitions unique to a single array instance. Any given column appears in each instance of the array. Using the example of employee dependents, the table structure looks like this:

Table 2. Structure of an array mapped to a Dependents table.

KEY	DEP_LAST_NAME	DEP_FIRST_NAME	DEP_GENDER	DEP_SSN	DEP_DOB

One common reason to create multiple table definitions for a database or file is that a table contains multiple, non-nested record arrays with a fixed number of instances. When the table has multiple arrays and instances, queries that reference the table yield result sets that are too large.

You can use an algorithm to calculate the number of rows in the result set from a query on a table that contain arrays, before any filtering predicates that are supplied on the WHERE clause. The number of rows in the result set is the Cartesian product of:

- The product of the instances in each record array
- The number of physical records

In this example, if:

1. The employee has:
   - Four dependents
   - Two emergency contacts
   - Three assignments
2. The database has ten employee records

Then a query generates `4 * 2 * 3 * 10 = 240` rows in the result set, and 24 rows for each employee record.

Another scenario is that a record has multiple arrays, and the last array has a variable length (in this example, the number of dependents is unknown). With variable-length arrays, you can't calculate the actual number of rows in the result set.

You cannot map columns that appear in a table subsequent to variable-length arrays at all, because these columns do not appear at a predictable offset.

Another common reason to create multiple table definitions for a database or file is that the database or file contains redefined information. The database record or segment consists of fixed information that includes a type code field that identifies the remaining contents of the record or segment.

For read-only queries, you can supply filtering information in a view definition that applications use when they need to access data for a given record type. You cannot perform updates against a view, however. If an application needs to update a database or file containing redefined information, then the application must use the base table name with WHERE filtering.

To query redefined information, define multiple logical tables — one for each type of record.

1. Each table must contain columns that identify common key information and a column for the type code field.
2. These columns are followed by type-specific columns.
3. When accessing these tables, do one of the following:
   - Supply a WHERE clause to filter the records based on the type code value.
   - Create a view with the WHERE clause and query the view.

Change capture

You cannot map *record array definitions* for change capture. If you want to map *record arrays* for change capture, map the columns in a flattened structure that provides a separate column for each array instance and field. For example, say you map a record array for employee dependents that contains five fields:

1. DEP_LAST_NAME

2. DEP_FIRST_NAME
3. DEP_GENDER
4. DEP_SSN
5. DEP_DOB

If you want to support up to ten dependents, you must map 50 columns with names that uniquely identify each instance and field. In this example, the column names range from DEP_LAST_NAME_1 to DEP_DOB_50.

Change capture does not currently support record array definitions, because change capture must send one notification per change. In this example, a message describing an update to a single field in the employee table inserts ten rows of changed data in the logical table on the data server — one for each dependent.

Another unsupported scenario is adding a new dependent to an array that uses a DEPENDING ON or similar clause. In this example, database operations calculate the number of dependents on the fly, based on the value of a column that stores the number of dependents, such as EMPL_DEP_COUNT. Adding a new dependent *updates* a single record in the source database, but the change *inserts* a new row into the logical table on the data server.

Flatten the structure of your logical table to avoid the confusing insert and update logic in scenarios like those described here. Event publishing applications that consume the data cannot process the logic.

PRIMARY KEY *column-name*

Identifies the columns that uniquely identify an IMS database record and the segment instances that this table references in an IMS database.

The specification of primary key information is always valid for a table that references an IMS database.

The ability to determine what constitutes a good set of primary key columns versus a bad set of columns depends primarily upon whether *perceived-root-segment-name* is explicitly specified or implicitly identified, for example by the root segment.

The primary key does not enforce the same restrictions that the CREATE INDEX statement does. There is no prohibition against identifying the columns that make up the primary key sequence field or a DEDB, HDAM or PHDAM database as primary key columns. Likewise, you can use primary key columns for child segments. References to columns that map to the sequence fields of all of the child segments in the database hierarchy are also good primary key column references. Under the assumption that the fully concatenated key is unique, references to child segment sequence fields that are not unique are also acceptable . A warning message is generated when a reference is made to a non-unique child sequence field.

Example

The following is an example of a CREATE TABLE statement for IMS.

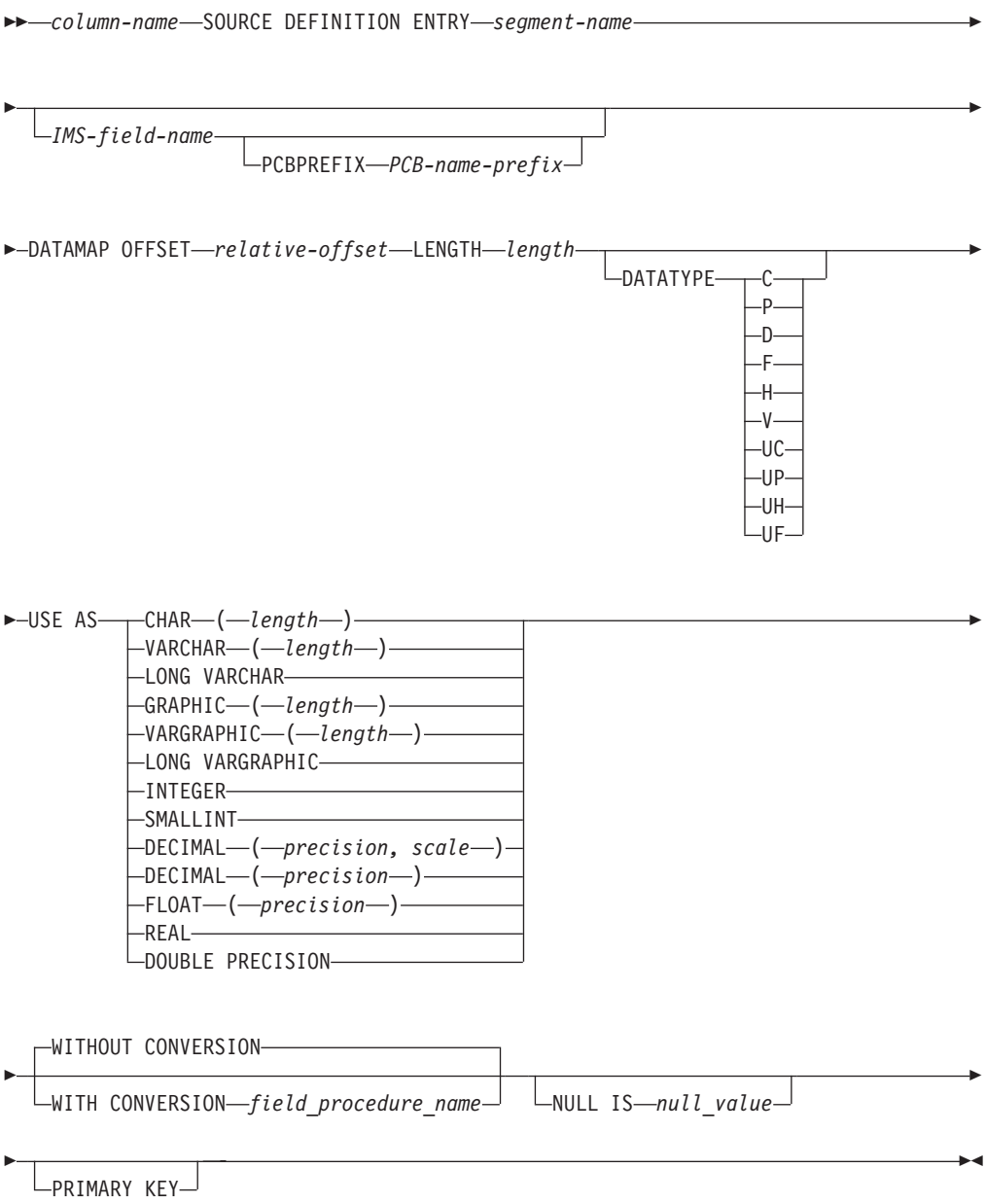
```
CREATE TABLE CXAIMS.IMSALDB DBTYPE IMS
  FVT52901 INDEXROOT FVTROOT FVTROOT
  SCHEDULEPSB(PF52901U) PCBPREFIX FVT
(
 ALDBIMSKEY  SOURCE DEFINITION ENTRY FVTROOT
```

```
   DATAMAP OFFSET 0 LENGTH 4 DATATYPE F
   USE AS INTEGER,
  ALDBIMSCHAR  SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 4 LENGTH 254 DATATYPE C
   USE AS CHAR(254)
   NULL IS X'4040',

  ALDBIMSLDECIMAL SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 266 LENGTH  8 DATATYPE P
   USE AS DECIMAL(15,0)
   NULL IS X'000000000000000C',
  ALDBIMSDECIMALMAX SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 282 LENGTH  8 DATATYPE P
   USE AS DECIMAL(15,15)
   NULL IS X'000000000000000C',
/*                                                 */
  ALDBIMSLFLOAT SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 294 LENGTH 8 DATATYPE D
   USE AS FLOAT(53)
   NULL IS X'0000',
  ALDBIMSDBLPREC SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 302 LENGTH 8 DATATYPE D
   USE AS FLOAT(53)
   NULL IS X'0000',
  ALDBIMSINTEGER SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 310 LENGTH 4 DATATYPE F
   USE AS INTEGER
   NULL IS X'00000000',
  ALDBIMSREAL SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 314 LENGTH 4 DATATYPE D
   USE AS FLOAT(21)
   NULL IS X'0000',
  ALDBIMSSMALLINT SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 318 LENGTH 2 DATATYPE H
   USE AS SMALLINT
   NULL IS X'0000',
  ALDBIMSVCHAR SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 320 LENGTH 256 DATATYPE V
   USE AS VARCHAR(254)
   NULL IS X'4040',
/*                                                 */
  ALDBIMSLVCHAR SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 576 LENGTH 1026 DATATYPE V
   USE AS VARCHAR(1026)
   NULL IS X'4040',
/*                                                 */

  ALDBIMSLVGRAPHIC SOURCE DEFINITION ENTRY FVTROOT
   DATAMAP OFFSET 2112 LENGTH 1025 DATATYPE V
   USE AS VARGRAPHIC(1025)
/* USE AS LONG VARGRAPHIC                           */
   NULL IS X'4040',
  PRIMARY KEY (ALDBIMSKEY)
 );
```

Columns for IMS

You can define a column that references an IMS database. For IMS column definitions, you must also identify the name of the segment where the column resides.

Syntax



Parameters

column-name
Identifies the name of the column.

Specifies the column name that is a long identifier. Column names cannot be qualified with a CREATE TABLE statement.

segment-name
Identifies the segment where the column is located. The name is a short native identifier.

The name must also exist in the DBD that *DBD_name* specifies and must exist in the hierarchic path between the *perceived_root_segment_name* identifier and the *leaf_segment_name* identifier.

DATAMAP
Specifies the relative offset for a column. If a LENGTH keyword is specified, information about the length of the column is also defined.

OFFSET *relative-offset*

Follows the DATAMAP keyword to set the offset for the column. For some data sources like Adabas and CA-IDMS, explicit offset and length information does not need to be supplied because the system data dictionaries can provide this information. In these cases, the column definition does not provide offset information. The column definition only identifies the element or field name that the column corresponds to in the data source data dictionary. If a column definition references a portion of a dictionary element or field, the grammar supports specifying column offset and length within the element or field. The column definition can identify the column offset of the start of the column and the length of the element or field that the column is being mapped to.

The relative offset identifies the relative zero offset of the starting position of the column within the object that the column is associated with. For simple objects like a VSAM or sequential file, the offset is generally measured from the start of the record. For more complex databases, like IMS or CA-IDMS, the relative offset is measured from the start of a segment or record. If the column is defined within a record array, then the relative offset is measured from the start of the record array that is measured from the start of the fragment that the column is associated with.

Note: Columns do not need to be defined in ascending relative offset starting sequence. When a mapping contains fixed length record arrays with additional columns following the record array, the starting offsets typically increase for the columns before the record array definition. For the columns in the record array, the relative offset information is reset to one and to larger numbers for those columns that exist after the record array.

LENGTH *length*
The LENGTH clause is required for IMS column definitions.

Specifies the length of the column. Generally, inconsistencies between the length that is specified using the LENGTH keyword and the length that is obtained from the SQL data type definition on the USE AS clause are ignored. For native DECIMAL data types the LENGTH must match the computed physical length of the column, based on the precision and scale specified in the USE AS clause when the USE AS precision is non-zero. For USE AS DECIMAL definitions the LENGTH must match the computed physical length of the column when the scale is non-zero and greater than the precision. Additionally, differences between the *length* and the SQL length specification for VARCHAR and VARGRAPHIC data types identify how to interpret the length attribute for the varying length column. The following rules are applied for VARCHAR data types:

- If the *length* and the VARCHAR lengths are identical, then the control length is assumed to include the length (2 bytes) that is taken up by the control length component.
- If the *length* is two bytes greater than the VARCHAR length, then the control length component is assumed to identify the actual length of the data.

For a varying length graphic string, the same kind of conventions are used. However, the lengths are expressed in DBCS characters. Therefore the rules are as follows:

- If the *length* and the VARGRAPHIC lengths are identical, then the control length is assumed to include the length (1 DBCS character, which is 2 bytes) that is taken up by the control length of a component.
- If the *length* is one character greater (two bytes) than the VARGRAPHIC length, then the control length component is assumed to identify the actual length of the data in characters.

Generally, the USE AS length must match the *length* value specified. However, for VARCHAR, the *length* can be two bytes off. For VARGRAPHIC, the length can be different by one. For these data types, the differences do not represent a conflict.

DATATYPE

Identifies the native format of the column.

The following table identifies the basic native data types.

Table 3. Native data types

DATATYPE value	Contents	Standard SQL data type	Other SQL data types
C	Mixed mode character data. When the SQL data type is DECIMAL, the data is assumed to consist wholly of numbers with the right most number identifying the sign.	CHAR	DECIMAL, VARCHAR, GRAPHIC, or VARGRAPHIC
P	Packed decimal data where the sign is stored in the far right aggregation of four bits.	DECIMAL	N/A
D[1]	Floating point data. The columns length or precision determines whether the SQL data type is REAL or DOUBLE PRECISION.	REAL or DOUBLE PRECISION	N/A
F	32 bit signed binary value where the sign is in the high order bit.	INTEGER	N/A
H	16 bit signed binary value where the sign is in the high order bit.	SMALLINT	N/A
V	Variable mixed mode character data, where the actual data is preceded by a 16 bit signed binary number that identifies the actual length of the data.	VARCHAR	LONG VARCHAR, VARGRAPHIC, or LONG VARGRAPHIC

Table 3. Native data types (continued)

DATATYPE value	Contents	Standard SQL data type	Other SQL data types
UC	Unsigned zoned decimal data where the last character does not identify the sign. The value is always a positive value.	DECIMAL	CHAR
UP	Packed decimal data where the sign nibble is ignored. The value is always positive.	DECIMAL	N/A
UF[2]	Unsigned 32 bit binary value.	INTEGER	N/A
UH[3]	Unsigned 16 bit binary value.	SMALLINT	N/A

[1]The SQL data type is identified as FLOAT, and the column length is either 4 or 8, based on the column precision. In the USE AS clause, these floating point data types can be identified as FLOAT(precision), REAL, or DOUBLE PRECISION. REAL is shorthand for a 4 byte floating point number. DOUBLE PRECISION is shorthand for an 8-byte floating point number. For FLOAT, the maximum precision is specified. If the value is in the range 1 21, the column represents a 4 byte floating point number. For precisions in the range 22 53, the column represents an 8 byte floating point number. The GUI does not need to specify a native data type for floating point columns because the data type is defaulted based on the USE AS clause.

[2]Specifying a DATATYPE of UF or UH has no effect on the contents or formatting of the underlying value that is stored in the database. Because the SQL INTEGER and SMALLINT data types are always signed the underlying binary value is always treated as a signed value. The UF and UH DATATYPE values are included in the list to be consistent with the UC and UP data types.

[3]Specifying a DATATYPE of UF or UH has no effect on the contents or formatting of the underlying value that is stored in the database. Because the SQL INTEGER and SMALLINT data types are always signed the underlying binary value is always treated as a signed value. The UF and UH DATATYPE values are included in the list to be consistent with the UC and UP data types.

Some database-specific data types are supported that are not shown in the above table.

If DATATYPE information is not specified, the native data type is synthesized based on the column of the SQL data type or from the database system.

The SQL data type information in the USE AS clause identifies the value in the SIGNED column in the following instances:

- One character code is supplied
- No DATATYPE information is specified
- Native data type information is not obtained from the database system

USE AS

Identifies the SQL data type for the column.

The following table describes the data types for columns. The non-null SQLTYPE identifies the data type in internal control blocks and diagnostic trace information.

Table 4. SQL data type descriptions

Keyword identifier	Description	Maximum length	SQLTYPE
CHAR(length)	Fixed-length character string that contains mixed mode data.	255	452
VARCHAR(length)	Variable length character string that contains mixed mode data. A half-word length component precedes the character string and identifies the actual length of the data. The length field does not include the length of the length field.	32704	448
LONG VARCHAR	Long character string that contains mixed mode data. A half-word length component precedes the character string and identifies the actual length of the data. The length field does not include the length of the length field.	32704	456
GRAPHIC(length)	Fixed-length graphic string that is assumed to contain pure DBCS data without shift codes. The length is expressed in DBCS characters, and not bytes.	127	468
VARGRAPHIC(length)	Varying-length graphic string that is assumed to contain pure DBCS data without shift codes. A half-word length component precedes the graphic string and identifies the actual length of the data. The length is expressed in DBCS characters, and not bytes.	16352	464

Table 4. SQL data type descriptions (continued)

Keyword identifier	Description	Maximum length	SQLTYPE
LONG VARGRAPHIC	Long graphic string that is assumed to contain pure DBCS data without shift codes. A half-word length component precedes the graphic string and identifies the actual length of the data. The length is expressed in DBCS characters, and not bytes.	16352	472
INTEGER	Large integer.	Exactly 4	496
SMALLINT	Small integer.	Exactly 2	500
DECIMAL (precision,scale) or DECIMAL(precision)	Packed decimal data. A valid precision value is between 1 and 31. A valid scale value is between zero and the precision value.	31	484
FLOAT(precision)	Floating point number. Depending upon the precision, a column with a FLOAT data type takes on the attributes of a REAL or DOUBLE PRECISION data type. When precision is in the range of 1 to 21, the column is treated as a REAL. For precisions between 22 and 53, the column takes on DOUBLE PRECISION attributes. A precision of zero or greater than 53 is nonvalid.	4 or 8	480
REAL	Floating point number with a range of magnitude of approximately 5.4E 79 to 7.2E+75.	4	480
DOUBLE PRECISION	Floating pointer with a range of magnitude of approximately 5.4E 79 to 7.2E+75.	8	480

When you use a USE AS LONG VARCHAR or USE AS LONG VARGRAPHIC clause, you do not specify a length for the column. The length is based on the

physical attributes that are obtained about the object record or segment where the column is. When you use a USE AS VARCHAR clause with a length greater than 254, that column is changed into a LONG VARCHAR column. The length in the clause is discarded. The maximum physical length is computed based on physical attributes. The same behavior occurs for a USE AS VARGRAPHIC clause when the length is greater than 127.

WITHOUT CONVERSION
Specifies that a field procedure does not exist for the column.

WITH CONVERSION
Identifies the name of the load module for a field procedure, if a field procedure is required. The field procedure name is a short identifier.

NULL IS *null_value*
Specifies a value that identifies when the contents of the column is null. By default, the data in a column never contains a null value. This situation is true even for variable length character or graphic columns where the length component indicates that there is no data in the variable length column.

A null value is a character string that can specify up to 16 characters of data. Specifies the value in hexadecimal format. The total length of the string is 35 characters.

A column contains null values based on *null-value*. If the start of the column data matches the null value, then that instance of the column is null. For example, if a column is defined as CHAR(10) and a null value of x'4040' is specified, the null value length is 2. If the first 2 bytes of the column contain spaces, the column is also null.

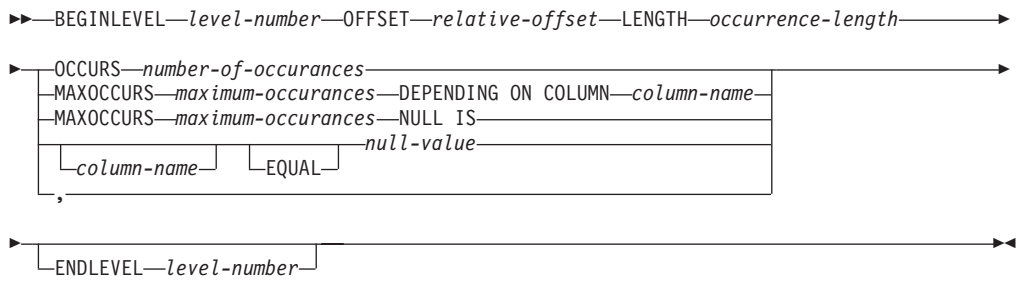
PRIMARY KEY
Identifies the column to be one of the columns that constitute the primary key for the table. This form of primary key identification is mutually exclusive with specifying a primary key at the end of the CREATE TABLE statement.

You can identify a primary key at the column level when the columns that make up the primary key for the table are defined in ordinal sequence within the table definition.

Record arrays for IMS

Record array definitions are used to identify and handle data that can repeat multiple times within a non-DB2 database or file system.

Syntax



Parameters

BEGINLEVEL *level-number*

Identifies the start of a record array.

The *level-number* on a BEGINLEVEL or ENDLEVEL definition identifies the nesting level of the record array within the parent fragment. Multiple record arrays can exist with the same *level-number*. However, these situations within a single table represent a questionable mapping. Ideally you should create separate tables for each record array that exists at the same level. The reason is that when accessing these kinds of tables, for a single database access, the query processor can generate the Cartesian product for the maximum occurrences of all record arrays that are defined.

The following example shows a multiple record array. In this example, the EMPL-ADDRESS data item occurs three times and the EMPL-DEPENDENTS, EMPL-DEP-NAME and EMPL-DEP-DOB data items occur ten times. By default the query processor generates 30 rows for each EMPL-RECORD read.

```
01 EMPL-RECORD
   05 EMPL-SSN               PIC 9(9)
   05 EMPL_NAME
     10  EMPL-LAST           PIC X(30)
     10  EMPL_FIRST          PIC X(30)
     10  EMPL_MI             PIC X
   05 EMPL-ADDRESS           PIC X(30) OCCURS 3 TIMES
   05 EMPL-DEPENDENTS        OCCURS 10 TIMES
     10  EMPL-DEP-NAME       PIC X(30)
     10  EMPL-DEP-DOB        PIC 9(8)
```

Figure 6. Example of a multiple record array

If the information in the above example needs to be accessed in a single table definition, then two record array definitions are required. These record arrays are both level number 1 arrays because the employee address information exists first and repeats three times followed by the employee dependent information that occurs ten times.

OFFSET *relative-offset*

The relative offset identifies the relative starting position of the record array within the segment or within a parent record array definition.

The relative offset is a required numeric parameter.

LENGTH *occurrence-length*

Iidentifies the length in bytes of one occurrence of the repeating data.

MAXOCCURS *maximum-occurrences* **DEPENDING ON COLUMN** *column-name*

Identifies a record array that occurs a variable number of times. The number of actual instances that exist in a particular database record is identified by the contents of a column definition that exists before the record array definition.

The following example uses an OCCURS DEPENDING ON COLUMN clause that defines a variably occurring record array. In this example, the EMPL-DEP-COUNT data item identifies the number of dependents that exist in the EMPL-DEPENDENTS record array.

```
01 EMPL-RECORD
   05 EMPL-SSN               PIC 9(9)
   05 EMPL_NAME
     10  EMPL-LAST           PIC X(30)
     10  EMPL_FIRST          PIC X(30)
     10  EMPL_MI             PIC X
   05 EMPL-ADDRESS           PIC X(30) OCCURS 3 TIMES
   05 EMPL-DEP-COUNT         PIC S9(4) COMP
   05 EMPL-DEPENDENTS        OCCURS 10 TIMES
                             DEPENDING ON EMPL-DEP-COUNT
     10  EMPL-DEP-NAME       PIC X(30)
     10  EMPL-DEP-DOB        PIC 9(8)
```

Figure 7. Example of an OCCURS DEPENDING ON defintion

You cannot map columns that occur after a variable length array, because the starting offsets of data items that are defined after an OCCURS DEPENDING ON definition are not at a fixed offset. The starting offsets float from one record instance to the next, based on the actual number of occurrences that are identified by the contents of the DEPENDING ON data item. The Classic federation servers do not currently support calculated field offsets, and you cannot access these fields.

The *maximum-occurrences* value must be a numeric value, and *column-name* must specify the name of a column in the table. The control column must contain a numeric value. The query process supports SQL types of SMALLINT, INTEGER, DECIMAL, or CHAR (if the column contains zoned decimal data).

MAXOCCURS *maximum-occurrences* **NULL IS**

Defines a record array that repeats a fixed number of times. You specify a value that denotes a null value. When a column in the record array contains the specified value, the content of the column is interpreted as null.

The *maximum-occurrences* value must be numeric. The *null-value* is a character string of up to 32 characters. The *null-value* can also be a 67-character hexadecimal string that consists of 64 hexadecimal nibbles, the leading x, and the surrounding quotation marks.

The minimal definition for this kind of array structure has the NULL IS keyword followed by the *null-value* string. With this syntax, a record array element contains null values based on the length of the *null-value* string. If the start of the column data matches the *null-value* then that instance of the repeating group of columns or record arrays is null. For example, a null-value of x'4040' is specified; the null value length is 2 and if the first two bytes of a record entry match the null value the entry is null. Processing of the array contents is skipped and the next element in the record array is inspected.

Elements or the array are processed like this:

1. The first element in the record array is located based on the relative-offset information.

   The physical location is computed based on the parent fragment offset information.
2. The second through *nth* instances starting locations are determined by incrementing the physical location by the occurrence length value.

Alternatively, you can check the entire contents of a record array to determine if it contains a null value by using the ALL keyword. For example, the syntax is NULL IS ALL null-value. With this form, only a single character of null value data can be supplied in either character or hexadecimal format.

When you add *column-name* EQUAL to the NULL IS clause, the null checking for a record array element is confined to the contents of a single column. Ideally, the column needs to exist within the record array. However, that column is not a requirement.

When column-level null checking is requested, the same kinds of options are available as with entry-level checking. The same kinds of errors can be generated. If the ALL keyword is not specified, then the contents of the column in each record array entry is checked to see if it matches the supplied value. If so, the entry is null and its contents are ignored. Likewise if the ALL keyword is supplied, only a single character null value can be specified. The entire contents of the column are checked to see if a record array instance is null and therefore ignored.

ENDLEVEL *level-number*
Identifies the end of the definition.

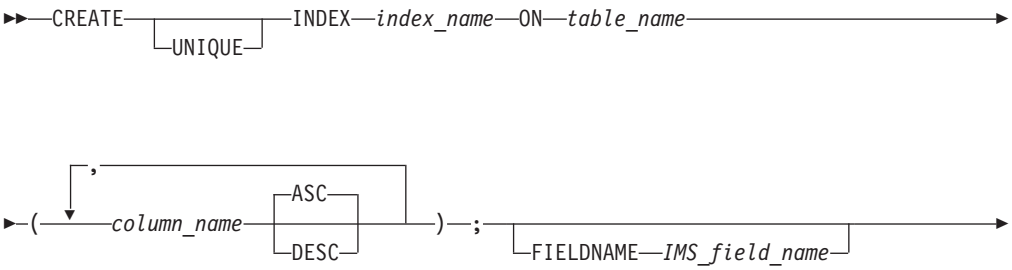
CREATE INDEX statement for IMS

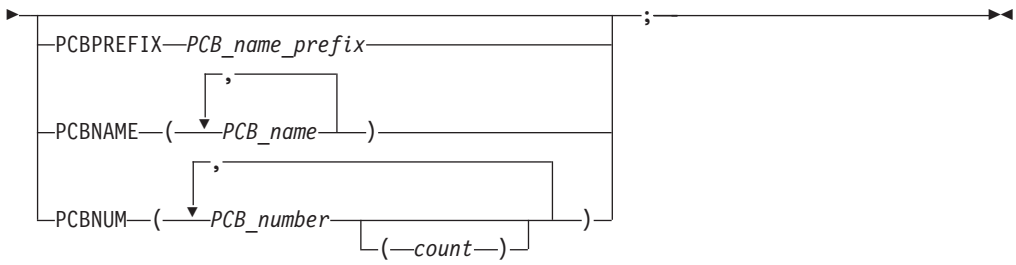
You can use the CREATE INDEX statement to define an index that references the columns that make up the IMS primary key (FIELD). You can also define an index that references the columns that map to the SRCH fields in the XDFLD statement of a secondary index definition to access the IMS database.

Indexes identify columns in a table that correspond to a physical index that is in the source database. The query processor uses the contents of the columns that are referenced in a WHERE clause to create the index. The query processor attempts to build either a full key value to use in database access or a partial key value. The partial key can be used to perform a range scan against the target database to reduce the number of records that are accessed.

Although you might not be able to define an index against columns that correspond to the primary key, you can identify these columns as primary key columns. This primary key information does not affect existing connector index selection and access optimization. The primary key information is made available for use by front-end tools and for replication purposes, where key information is either required or beneficial.

Syntax





Parameters

CREATE INDEX *index_name*

Identifies the SQL statement as an index definition statement. A unique index does not have any restrictions about the columns that make up the index. For example, a unique index column can contain null values.

If qualified, the index name is a two-part name, and the authorization ID that qualifies the name is the owner of the index. If an unqualified table name is supplied, the owner name is the authorization ID from the CURRENT SQLID special register.

ON *table_name*

Identifies the table for which the index is being defined. The table name can be a qualified or unqualified table name. For unqualified names, the table owner is from the CURRENT SQLID special register.

The table name is validated with the standard syntax checks that are associated with identifiers, and then the existence of the table is verified. Do not define an index on a view.

column_name **ASC/DESC**

Specifies column names that make up the index key. Optionally, you can identify whether the column is stored or accessed in ascending (ASC) or descending (DESC) key sequence. By default, the key is in ascending key sequence.

There are no fixed limits on the number of columns that you can identify as key columns for an index. The only requirements are as follows:

- The column must exist in the table.
- The column must map to what constitutes a key in the target database. In most cases, this determination is based on the starting offset and length of the column as compared to the starting offset and length of the key in the target database.
- The column cannot overlap another column within the key definition. This determination is based on the starting offset and length of a column as compared to the starting offsets and length of the other columns that are identified as key columns for the index definition.
- For the key column, generally varying length and graphic data types are not supported.

Any kind of varying-length character or graphic string is not allowed.
If the CREATE TABLE statement specifies a segment for the INDEXROOT keyword and that segment is not a root segment, the key columns can only reference an XDFLD that is defined by INDEXROOT. The key columns

reference offsets and lengths in the source segment that the XDFLD references. If INDEXROOT is not specified in the CREATE TABLE statement or the segment is the root segment for the DBD, key columns can be specified for any XDFLD and can reference columns that map to the primary key sequence field of the database.
If *DBD_name* in the CREATE TABLE statement references a logical DBD, the key columns can reference an XDFLD in either of the physical segments that are referenced by the INDEXROOT segment. However, if INDEXROOT is not specified in the CREATE TABLE statement or the segment is the root segment for the DBD, primary key references are supported only for the first physical database that is referenced by the logical DBD.
If the index is defined against the root segment of the database and the DBD_NAME in the CREATE TABLE statement is a DEDB, HDAM or PHDAM database, the index cannot consist of key columns that reference the primary key sequence field of the IMS database. This also holds true if the DBD_NAME in the CREATE TABLE statement references a logical database and the root segment of the first physical database is a DEDB, HDAM or PHDAM database

PCBPREFIX *PCB_name_prefix*
Identifies the PCBs that access the IMS database when the index is selected based on the WHERE clause. *PCB_name prefix* is 1 to 7 characters in length and follows IMS PCB naming conventions.

If a PCBPREFIX is in the IMS CREATE TABLE statement, you must specify a PCBPREFIX in the CREATE INDEX statement. When the index accesses the table, the index-level PCBPREFIX is used in preference to any PCB prefix information at the table level.

If the index corresponds to the primary key sequence field of the IMS database, the IMS PCB definitions that correspond to the PCBPREFIX information must not contain a PROCSEQ definition.

Also, if the index definition corresponds to an XDFLD definition, the PCBs must contain a PROCSEQ. This process identifies the secondary index DBD that corresponds to the XDFLD that the index key columns match.

PCBNAME (*PCB_name*,...)
Specifies up to 5 PCBs that access an IMS database through a table. Multiple PCBs are required if the same table is referenced more than once in an SQL statement, or when the same PCB is associated with more than one table and these additional tables are referenced in a single SQL statement.

Each PCB name in the list can be up to 8 characters long. For multiple PCB names, separate each name by a comma.

PCBNUM (*PCB_number* (*count*),...)
Specifies a number of PCBs to access the IMS database for the table. Multiple PCBs are required under either of these conditions:

- The same table is referenced more than once in an SQL statement.
- The same PSB is associated with more than one table, the PCBs in the PSB have sensitivity to the segments that the table is accessing, the same PCB ordinal numbers are specified for these tables, and these additional tables are referenced in a single SQL statement.

You can specify up to ten sets of PCB number ranges. These PCB numbers can be listed in any order and represent the order in which a PSB is checked to determine whether a PCB is used to access the IMS database.

For each item in the list, two different formats are used to identify the PCB numbers to be checked. The simplest method is to identify the PCB numbers to

be checked separated by commas. The second technique identifies a range specification that consists of a starting PCB number that is followed by parenthesis and a number that identifies the number of PCBs to be checked from the starting number.

For either technique, the PCB number represents the relative 1 ordinal number of the PCB that is to be checked. Because an I/O PCB needs to be defined in each PSB that Classic Federation uses, the minimum practical PCB number that can be specified starts at two.

Example

The following is an example of a CREATE INDEX statement for IMS.

```
CREATE UNIQUE INDEX CXAIMS.IMSALDB_IDX1 ON CXAIMS.IMSALDB (ALDBIMSKEY ASC);
```

ALTER TABLE statement for IMS

You can use the ALTER TABLE statement to create tables that reference an IMS database.

Authorization

You must have the one of the following authorities to run the ALTER TABLE statement:

- SYSADM
- DBADM for the database type that is referenced in the DBNAME column for the table being altered
- Ownership of the table being altered

Syntax

Parameters

table-name
Identifies the table for which the DATA CAPTURE flag is being altered. The table name can be a qualified or unqualified table name. If an unqualified name is supplied, the table owner is from the CURRENT SQLID special register.

DATA CAPTURE

Indicates that the ALTER TABLE statement is setting the value of the DATA CAPTURE flag.

CHANGES
Allows a change-capture agent to capture changes to the non-relational data that the table is mapped to.

NONE
Disallows change-capture agents to capture changes to the non-relational data that the table is mapped to.

Example

The following is an example of an ALTER TABLE statement for IMS.

```
ALTER TABLE CXAIMS.IMSALDB DATA CAPTURE CHANGES;
```

Chapter 3. VSAM

You can use the CREATE TABLE and ALTER TABLE statements to define tables that reference VSAM files. You can use the CREATE INDEX statement to define indexes that reference VSAM files.

CREATE TABLE statement for VSAM

You can use the CREATE TABLE statement to define a logical table that references a VSAM file.

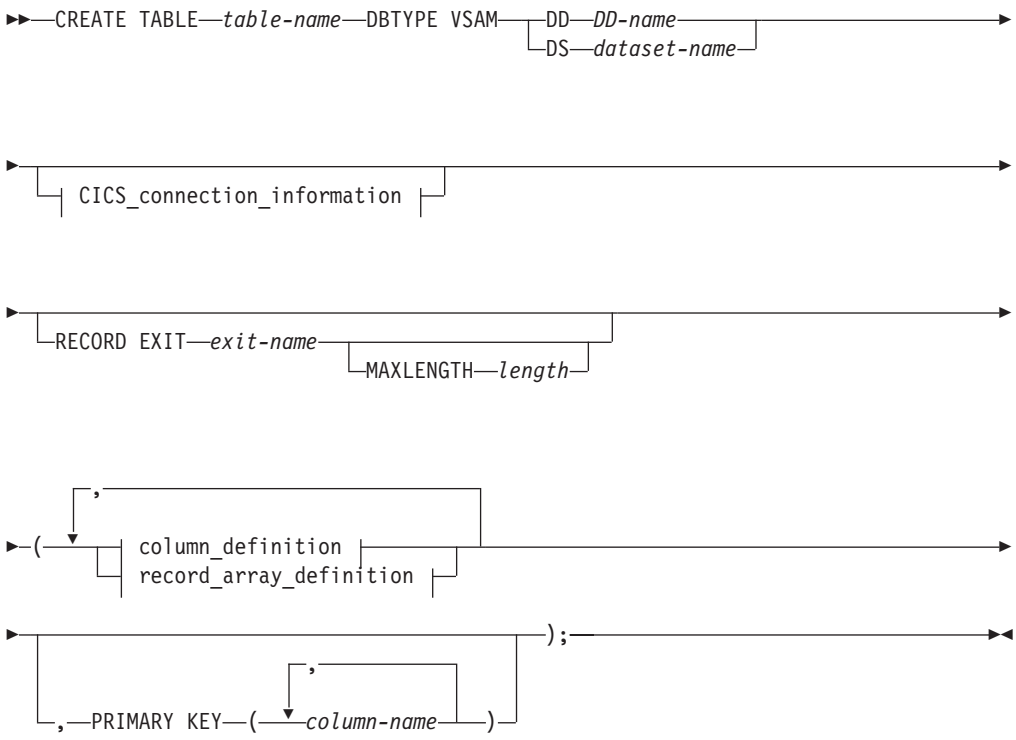
Authorization

The connected user ID must have one of the following privileges to run the CREATE TABLE statement:

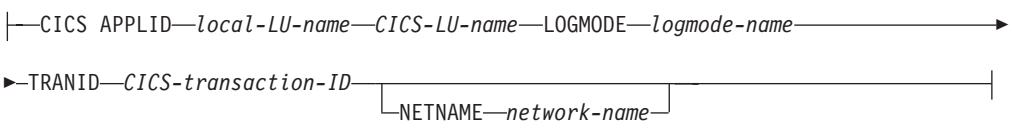
- SYSADM
- DBADM for the database type that is referenced in the DBTYPE clause

The owner has all table privileges on the table (such as SELECT, UPDATE, and so on) and the authority to drop the table. The owner can grant equivalent use privileges on the table.

Syntax



CICS_connection_information:



Parameters

authorization-ID.table-name

Identifies the owner of the table and the name of the table that you want to create.

If you do not provide an authorization ID, the ID in the CURRENT SQLID special register is used.

You must create more than one table if you map to a database or a file that meets either of these criteria:

- The database or file contains repeating data.
- The database or file contains information about one or more distinct sub-objects, because the database or file is not designed to follow the third normalization rules, which are part of the standards to eliminate redundancies and inconsistencies in table data. In a table designed according to third normalization rules, each non-key column is independent of other non-key columns, and is dependent only upon the key.

DBTYPE VSAM

Specifies that the CREATE TABLE statement defines a logical table that references a VSAM file.

DD *DD-name*

Specifies the VSAM file that the table maps to. Specifies either a local file reference or a reference to a CICS file definition table (FDT) entry name.

If *DD-name* represents a local file reference, a DD clause that references the VSAM file must exist in the Classic federation data server JCL.

If *DD-name* references an FDT entry name, you must specify CICS connection information with the CICS_connection_information clause.

DD-name must correspond to a cluster or alternate index path definition for a VSAM ESDS, KSDS or RRDS data set.

DD-name is a short native identifier that conforms to the naming conventions for a z/OS JCL DD statement.

DS *data-set-name*

Specifies the VSAM file that the table definition accesses. Designates a VSAM cluster definition or a PATH component when the VSAM file is accessed from a VSAM alternate index.

The data set name must correspond to a cluster or alternate index path definition for a VSAM ESDS, KSDS or RRDS data set.

The name can be 1 to 44 characters in length and follows z/OS naming conventions for VSAM data sets.

CICS_connection_information

Specifies CICS connection information that is required only if your VSAM files are managed by CICS.

For Classic federation, you must specify CICS connection information that identifies the CICS subsystem where the VSAM file is located.

For Classic event publishing or replication, you must specify CICS connection information to allow validation processing and to collect information about file attributes.

CICS APPLID
Specifies the names of two logical units (LUs) that establish communication with the target CICS subsystem.

local-LU-name
This value identifies the name of a local LU that is used to communicate with CICS. This identifier is one to eight characters in length. *local-LU-name* corresponds to either the ACBNAME or the name (label) on the VTAM APPL definition. *local-LU-name* must be active on the image where the server runs.

The name follows VTAM naming conventions. The SASCSAMP member CACCAPPL provides the sample local LU definitions. The sample names for *local-LU-name* are CACCICS1 and CACCICS2, which you can modify. Define the name to CICS as a CONNECTION definition. Sample connection definitions for CACCICS1 and CACCICS2 are in SASCSAMP member CACCDEF.

CICS-LU-name
Designates the VTAM LU 6.2 definition that a CICS region listens on for connection requests. This LU name is 1 to 8 characters in length. The name corresponds to the value of the APPLID parameter in the system initialization definition (DFHSIT macro) of the target CICS subsystem where the VSAM file is located. The name follows VTAM naming conventions.

LOGMODE *logmode-name*
Identifies the name of the VTAM log mode table that controls session establishment and the service-level properties of the session that CICS determines. The name is a short native identifier that designates the name of the VTAM logon-mode table that controls the session parameters for the conversation that is established between the local LU and the CICS LU. This name is 1 to 8 characters in length. The logon-mode table name corresponds to a z/OS load module that is accessible to VTAM. The definition for a Classic logon-mode table is in SASCSAMP member CACCMODE.

TRANID *CICS-transaction-ID*
Performs the following functions:

- Identifies the name of the Classic-supplied transaction that verifies the existence of the VSAM file

The name of the CICS transaction is used for data access and validation purposes. This name is 1 to 4 characters in length. In prior releases, *CICS-transaction-ID* was for data access only, and a separate transaction performed validation checking. In version 9.1, these two CICS transactions are combined. *CICS-transaction-ID* corresponds to the CICS TRANSACTION definition.

The SCACSAMP member CACCDEFS provides sample CICS transaction, connection, program, and session definitions. The sample *CICS-transaction-ID* identifier is EXV1, which you can modify.

NETNAME *network-name*

Enables the federation data server to communicate with a remote CICS system by identifying the name of the network where the CICS subsystem is running.

The name of the network where *CICS-LU-name* resides corresponds to the CICS subsystem that accesses a VSAM file. This identifier is 1 to 8 characters in length.

The name is identified on the NETWORK VTAM macro definition on the local image to identify the remote SNA network where the CICS subsystem resides.

The name follows VTAM naming conventions.

RECORD EXIT *exit-name*

Identifies that a record processing exit is invoked for post processing, after a record is retrieved from the VSAM file. The record processing exit is also called for preprocessing before a VSAM record is inserted or updated. The record exit is used for altering the record contents or filtering specified records.

The record processing exit is not used in change capture.

The name is a short identifier that follows the naming conventions for a z/OS load module.

column_definition

Provides SQL descriptions of the contents of the VSAM file. Column mapping must be based on COBOL copybook definitions.

A table must contain at least one column and can contain up to 5,000 columns. Columns are named, and each name must be unique within the table.

record_array_definition

Identifies repeating data. Record array definitions contain column definitions and possibly more record array definitions.

Federated queries can use *record array definitions* if you create a separate table for each array. Change capture can use *record arrays* if you map the columns in a flattened structure that provides a separate column for each array instance and field.

Federated queries

For federated queries, define multiple tables. Each table consists of a single record array definition that contains the column definitions unique to a single array instance. Any given column appears in each instance of the array. Using the example of employee dependents, the table structure looks like this:

Table 5. Structure of an array mapped to a Dependents table.

KEY	DEP_LAST_NAME	DEP_FIRST_NAME	DEP_GENDER	DEP_SSN	DEP_DOB

One common reason to create multiple table definitions for a database or file is that a table contains multiple, non-nested record arrays with a fixed number of instances. When the table has multiple arrays and instances, queries that reference the table yield result sets that are too large.

You can use an algorithm to calculate the number of rows in the result set from a query on a table that contain arrays, before any filtering predicates that are supplied on the WHERE clause. The number of rows in the result set is the Cartesian product of:

- The product of the instances in each record array
- The number of physical records

In this example, if:

1. The employee has:
   - Four dependents
   - Two emergency contacts
   - Three assignments
2. The database has ten employee records

Then a query generates `4 * 2 * 3 * 10 = 240` rows in the result set, and 24 rows for each employee record.

Another scenario is that a record has multiple arrays, and the last array has a variable length (in this example, the number of dependents is unknown). With variable-length arrays, you can't calculate the actual number of rows in the result set.

You cannot map columns that appear in a table subsequent to variable-length arrays at all, because these columns do not appear at a predictable offset.

Another common reason to create multiple table definitions for a database or file is that the database or file contains redefined information. The database record or segment consists of fixed information that includes a type code field that identifies the remaining contents of the record or segment.

For read-only queries, you can supply filtering information in a view definition that applications use when they need to access data for a given record type. You cannot perform updates against a view, however. If an application needs to update a database or file containing redefined information, then the application must use the base table name with WHERE filtering.

To query redefined information, define multiple logical tables — one for each type of record.

1. Each table must contain columns that identify common key information and a column for the type code field.
2. These columns are followed by type-specific columns.
3. When accessing these tables, do one of the following:
   - Supply a WHERE clause to filter the records based on the type code value.
   - Create a view with the WHERE clause and query the view.

Change capture

You cannot map *record array definitions* for change capture. If you want to map *record arrays* for change capture, map the columns in a flattened structure that provides a separate column for each array instance and field. For example, say you map a record array for employee dependents that contains five fields:

1. DEP_LAST_NAME

2. DEP_FIRST_NAME
3. DEP_GENDER
4. DEP_SSN
5. DEP_DOB

If you want to support up to ten dependents, you must map 50 columns with names that uniquely identify each instance and field. In this example, the column names range from DEP_LAST_NAME_1 to DEP_DOB_50.

Change capture does not currently support record array definitions, because change capture must send one notification per change. In this example, a message describing an update to a single field in the employee table inserts ten rows of changed data in the logical table on the data server — one for each dependent.

Another unsupported scenario is adding a new dependent to an array that uses a DEPENDING ON or similar clause. In this example, database operations calculate the number of dependents on the fly, based on the value of a column that stores the number of dependents, such as EMPL_DEP_COUNT. Adding a new dependent *updates* a single record in the source database, but the change *inserts* a new row into the logical table on the data server.

Flatten the structure of your logical table to avoid the confusing insert and update logic in scenarios like those described here. Event publishing applications that consume the data cannot process the logic.

PRIMARY KEY *column-name*
Specifies the column names that uniquely identify each record in the VSAM file. Depending upon the type of VSAM file that the CREATE TABLE references, statement specification of primary key columns might not be appropriate.

Corresponds to what constitutes the target database or file system equivalent of a primary key. The columns might represent the set of columns that logically identify a unique row. If there are multiple ways to identify a unique row, then multiple tables should be created. Each table would have a different primary key made up of the set of columns that uniquely identifies each record.
You can use either of two approaches to define a primary key. Use the following criteria to determine whether to specify the PRIMARY KEY clause on the column definition or the CREATE TABLE statement.

- If the columns that make up a composite key are defined in the same sequence as represented by the actual primary key, PRIMARY KEY can be specified on the column definitions. When this form of primary key identification is used, the sequence in which the columns are identified represents the ordinal position that you must use to construct a composite key that uniquely identifies a row in the table.
- When PRIMARY KEY information is specified on the main CREATE TABLE statement, the columns do not need to be defined in ordinal sequence.

Example

The following is an example of a CREATE TABLE statement for VSAM that change capture cannot use. This table includes an array to capture data about employee dependents, but the number of array occurrences depends on the number of

dependents, that is to say, the value of EMPL_DEP_COUNT. Change capture cannot use arrays of this type. An ALTER statement on the table will fail.

```
CREATE TABLE "DBA"."EMPLOYEE" DBTYPE VSAM
 DS "SAMPLE.VSAM.EMPLOYEE"
 (

 "EMPL_LAST_NAME" SOURCE DEFINITION
  DATAMAP OFFSET 0 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_FIRST_NAME" SOURCE DEFINITION
  DATAMAP OFFSET 20 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_GENDER" SOURCE DEFINITION
  DATAMAP OFFSET 40 LENGTH 1
  DATATYPE C
  USE AS CHAR(1),
 "EMPL_SSN" SOURCE DEFINITION
  DATAMAP OFFSET 41 LENGTH 9
  DATATYPE UC
  USE AS CHAR(9),
 "EMPL_DOB" SOURCE DEFINITION
  DATAMAP OFFSET 50 LENGTH 4
  DATATYPE UP
  USE AS DECIMAL(6 , 0),
 "EMPL_ADDRESS_1" SOURCE DEFINITION
  DATAMAP OFFSET 54 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_ADDRESS_2" SOURCE DEFINITION
  DATAMAP OFFSET 74 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_STATE" SOURCE DEFINITION
  DATAMAP OFFSET 96 LENGTH 2
  DATATYPE C
  USE AS CHAR(2),
 "EMPL_ZIP" SOURCE DEFINITION
  DATAMAP OFFSET 98 LENGTH 5
  DATATYPE UC
  USE AS CHAR(5),
 "EMPL_DEP_COUNT" SOURCE DEFINITION
  DATAMAP OFFSET 103 LENGTH 2
  DATATYPE UH
  USE AS SMALLINT,
 "EMPL_COUNT" SOURCE DEFINITION
  DATAMAP OFFSET 105 LENGTH 2
  DATATYPE UH
  USE AS SMALLINT,
 BEGINLEVEL 1 OFFSET 107 LENGTH 54
  MAXOCCURS 20
  DEPENDING ON COLUMN "EMPL_DEP_COUNT",
 "DEP_LAST_NAME" SOURCE DEFINITION
  DATAMAP OFFSET 0 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "DEP_FIRST_NAME" SOURCE DEFINITION
  DATAMAP OFFSET 20 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "DEP_GENDER" SOURCE DEFINITION
  DATAMAP OFFSET 40 LENGTH 1
  DATATYPE C
  USE AS CHAR(1),
 "DEP_SSN" SOURCE DEFINITION
```

```
 DATAMAP OFFSET 41 LENGTH 9
 DATATYPE UC
 USE AS CHAR(9),
"DEP_DOB" SOURCE DEFINITION
 DATAMAP OFFSET 50 LENGTH 4
 DATATYPE UP
 USE AS DECIMAL(6 , 0),
ENDLEVEL 1);
```

The following is another example of a CREATE TABLE statement for VSAM, which creates an array that captures the same data about employee dependents as in the previous example. This table maps the array in a flattened structure that includes a column for each array instance. Change capture can use arrays of this type.

```
CREATE TABLE "DBA"."EMPLOYEE_EP" DBTYPE VSAM
 DS "SAMPLE.VSAM.EMPLOYEE"
 XM URL "XM1/DS1/QUE1"
 (

 "EMPL_LAST_NAME" SOURCE DEFINITION
  DATAMAP OFFSET 0 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_FIRST_NAME" SOURCE DEFINITION
  DATAMAP OFFSET 20 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_GENDER" SOURCE DEFINITION
  DATAMAP OFFSET 40 LENGTH 1
  DATATYPE C
  USE AS CHAR(1),
 "EMPL_SSN" SOURCE DEFINITION
  DATAMAP OFFSET 41 LENGTH 9
  DATATYPE UC
  USE AS CHAR(9),
 "EMPL_DOB" SOURCE DEFINITION
  DATAMAP OFFSET 50 LENGTH 4
  DATATYPE UP
  USE AS DECIMAL(6 , 0),
 "EMPL_ADDRESS_1" SOURCE DEFINITION
  DATAMAP OFFSET 54 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_ADDRESS_2" SOURCE DEFINITION
  DATAMAP OFFSET 74 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 "EMPL_STATE" SOURCE DEFINITION
  DATAMAP OFFSET 96 LENGTH 2
  DATATYPE C
  USE AS CHAR(2),
 "EMPL_ZIP" SOURCE DEFINITION
  DATAMAP OFFSET 98 LENGTH 5
  DATATYPE UC
  USE AS CHAR(5),
 "EMPL_DEP_COUNT" SOURCE DEFINITION
  DATAMAP OFFSET 103 LENGTH 2
  DATATYPE UH
  USE AS SMALLINT,
 "EMPL_COUNT" SOURCE DEFINITION
  DATAMAP OFFSET 105 LENGTH 2
  DATATYPE UH
  USE AS SMALLINT,
 "DEP_LAST_NAME_1" SOURCE DEFINITION
  DATAMAP OFFSET 107 LENGTH 20
  DATATYPE C
```

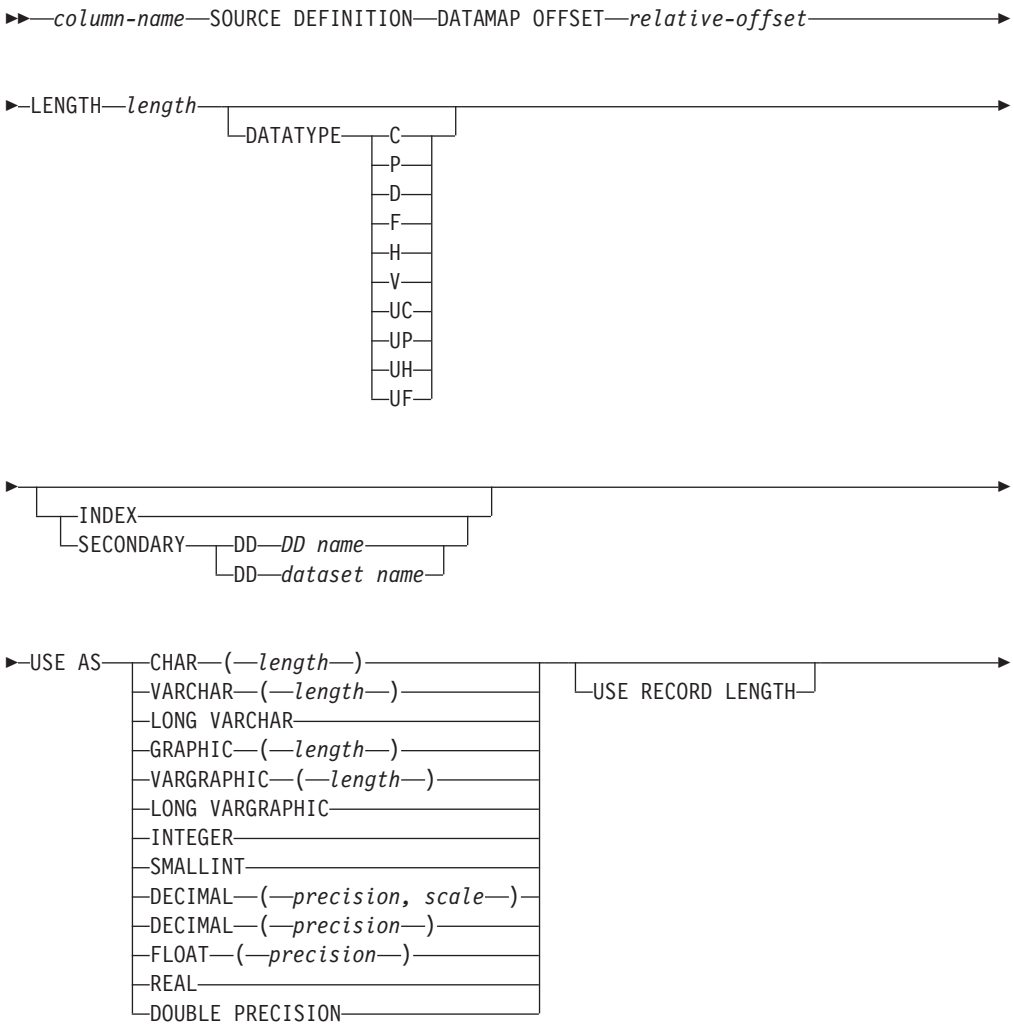
```
 USE AS CHAR(20),
"DEP_FIRST_NAME_1" SOURCE DEFINITION
 DATAMAP OFFSET 127 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_GENDER_1" SOURCE DEFINITION
 DATAMAP OFFSET 147 LENGTH 1
 DATATYPE C
 USE AS CHAR(1),
"DEP_SSN_1" SOURCE DEFINITION
 DATAMAP OFFSET 148 LENGTH 9
 DATATYPE UC
 USE AS CHAR(9),
"DEP_DOB_1" SOURCE DEFINITION
 DATAMAP OFFSET 157 LENGTH 4
 DATATYPE UP
 USE AS DECIMAL(6 , 0),
"DEP_LAST_NAME_2" SOURCE DEFINITION
 DATAMAP OFFSET 161 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_FIRST_NAME_2" SOURCE DEFINITION
 DATAMAP OFFSET 181 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_GENDER_2" SOURCE DEFINITION
 DATAMAP OFFSET 201 LENGTH 1
 DATATYPE C
 USE AS CHAR(1),
"DEP_SSN_2" SOURCE DEFINITION
 DATAMAP OFFSET 202 LENGTH 9
 DATATYPE UC
 USE AS CHAR(9),
"DEP_DOB_2" SOURCE DEFINITION
 DATAMAP OFFSET 211 LENGTH 4
 DATATYPE UP
 USE AS DECIMAL(6 , 0),
"DEP_LAST_NAME_3" SOURCE DEFINITION
 DATAMAP OFFSET 215 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_FIRST_NAME_3" SOURCE DEFINITION
 DATAMAP OFFSET 235 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_GENDER_3" SOURCE DEFINITION
 DATAMAP OFFSET 255 LENGTH 1
 DATATYPE C
 USE AS CHAR(1),
"DEP_SSN_3" SOURCE DEFINITION
 DATAMAP OFFSET 256 LENGTH 9
 DATATYPE UC
 USE AS CHAR(9),
"DEP_DOB_3" SOURCE DEFINITION
 DATAMAP OFFSET 265 LENGTH 4
 DATATYPE UP
 USE AS DECIMAL(6 , 0),
"DEP_LAST_NAME_4" SOURCE DEFINITION
 DATAMAP OFFSET 269 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_FIRST_NAME_4" SOURCE DEFINITION
 DATAMAP OFFSET 289 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
"DEP_GENDER_4" SOURCE DEFINITION
 DATAMAP OFFSET 309 LENGTH 1
```

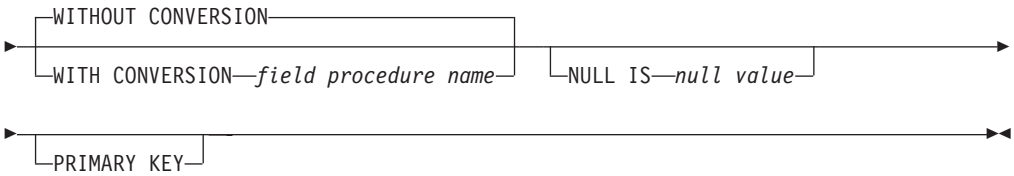
```
 DATATYPE C
 USE AS CHAR(1),
"DEP_SSN_4" SOURCE DEFINITION
 DATAMAP OFFSET 310 LENGTH 9
 DATATYPE UC
 USE AS CHAR(9),
"DEP_DOB_4" SOURCE DEFINITION
 DATAMAP OFFSET 319 LENGTH 4
 DATATYPE UP
 USE AS DECIMAL(6 , 0));
```

Columns for VSAM

Column definitions are a part of CREATE TABLE statements. You use column definitions to define the columns in a table that references a VSAM file. There are no differences between a sequential column definition and the generic column definitions.

Syntax





Parameters

column_name
Identifies the name of the column.

Specifies the column name that is a long identifier. Column names cannot be qualified with a CREATE TABLE statement.

DATAMAP
Specifies the relative offset for a column. If a LENGTH keyword is specified, information about the length of the column is also defined.

OFFSET *relative_offset*

Follows the DATAMAP keyword to set the offset for the column. For some data sources like Adabas and CA-IDMS, explicit offset and length information does not need to be supplied because the system data dictionaries can provide this information. In these cases, the column definition does not provide offset information. The column definition only identifies the element or field name that the column corresponds to in the data source data dictionary. If a column definition references a portion of a dictionary element or field, the grammar supports specifying column offset and length within the element or field. The column definition can identify the column offset of the start of the column and the length of the element or field that the column is being mapped to.

The relative offset identifies the relative zero offset of the starting position of the column within the object that the column is associated with. For simple objects like a VSAM or sequential file, the offset is generally measured from the start of the record. For more complex databases, like IMS or CA-IDMS, the relative offset is measured from the start of a segment or record. If the column is defined within a record array, then the relative offset is measured from the start of the record array that is measured from the start of the fragment that the column is associated with.

Note: Columns do not need to be defined in ascending relative offset starting sequence. When a mapping contains fixed length record arrays with additional columns following the record array, the starting offsets typically increase for the columns before the record array definition. For the columns in the record array, the relative offset information is reset to one and to larger numbers for those columns that exist after the record array.

LENGTH *length*

Specifies the length of the column. Generally, inconsistencies between the length that is specified using the LENGTH keyword and the length that is obtained from the SQL data type definition on the USE AS clause are ignored. For native DECIMAL data types the LENGTH must match the computed physical length of the column, based on the precision and scale specified in the USE AS clause when the USE AS precision is non-zero. For USE AS DECIMAL definitions the LENGTH must match the computed physical length of the column when the scale is non-zero and greater than the precision.

Additionally, differences between the *length* and the SQL length specification for VARCHAR and VARGRAPHIC data types identify how to interpret the length attribute for the varying length column. The following rules are applied for VARCHAR data types:

- If the *length* and the VARCHAR lengths are identical, then the control length is assumed to include the length (2 bytes) that is taken up by the control length component.
- If the *length* is two bytes greater than the VARCHAR length, then the control length component is assumed to identify the actual length of the data.

For a varying length graphic string, the same kind of conventions are used. However, the lengths are expressed in DBCS characters. Therefore the rules are as follows:

- If the *length* and the VARGRAPHIC lengths are identical, then the control length is assumed to include the length (1 DBCS character, which is 2 bytes) that is taken up by the control length of a component.
- If the *length* is one character greater (two bytes) than the VARGRAPHIC length, then the control length component is assumed to identify the actual length of the data in characters.

Generally, the USE AS length must match the *length* value specified. However, for VARCHAR, the *length* can be two bytes off. For VARGRAPHIC, the length can be different by one. For these data types, the differences do not represent a conflict.

DATATYPE

Identifies the native format of the column.

The following table identifies the basic native data types.

Table 6. Native data types

DATATYPE value	Contents	Standard SQL data type	Other SQL data types
C	Mixed mode character data. When the SQL data type is DECIMAL, the data is assumed to consist wholly of numbers with the right most number identifying the sign.	CHAR	DECIMAL, VARCHAR, GRAPHIC, or VARGRAPHIC
P	Packed decimal data where the sign is stored in the far right aggregation of four bits.	DECIMAL	N/A
D[1]	Floating point data. The columns length or precision determines whether the SQL data type is REAL or DOUBLE PRECISION.	REAL or DOUBLE PRECISION	N/A

Table 6. Native data types (continued)

DATATYPE value	Contents	Standard SQL data type	Other SQL data types
F	32 bit signed binary value where the sign is in the high order bit.	INTEGER	N/A
H	16 bit signed binary value where the sign is in the high order bit.	SMALLINT	N/A
V	Variable mixed mode character data, where the actual data is preceded by a 16 bit signed binary number that identifies the actual length of the data.	VARCHAR	LONG VARCHAR, VARGRAPHIC, or LONG VARGRAPHIC
UC	Unsigned zoned decimal data where the last character does not identify the sign. The value is always a positive value.	DECIMAL	CHAR
UP	Packed decimal data where the sign nibble is ignored. The value is always positive.	DECIMAL	N/A
UF[2]	Unsigned 32 bit binary value.	INTEGER	N/A
UH[3]	Unsigned 16 bit binary value.	SMALLINT	N/A

[1]The SQL data type is identified as FLOAT, and the column length is either 4 or 8, based on the column precision. In the USE AS clause, these floating point data types can be identified as FLOAT(precision), REAL, or DOUBLE PRECISION. REAL is shorthand for a 4 byte floating point number. DOUBLE PRECISION is shorthand for an 8-byte floating point number. For FLOAT, the maximum precision is specified. If the value is in the range 1 21, the column represents a 4 byte floating point number. For precisions in the range 22 53, the column represents an 8 byte floating point number. The GUI does not need to specify a native data type for floating point columns because the data type is defaulted based on the USE AS clause.

[2]Specifying a DATATYPE of UF or UH has no effect on the contents or formatting of the underlying value that is stored in the database. Because the SQL INTEGER and SMALLINT data types are always signed the underlying binary value is always treated as a signed value. The UF and UH DATATYPE values are included in the list to be consistent with the UC and UP data types.

[3]Specifying a DATATYPE of UF or UH has no effect on the contents or formatting of the underlying value that is stored in the database. Because the SQL INTEGER and SMALLINT data types are always signed the underlying binary value is always treated as a signed value. The UF and UH DATATYPE values are included in the list to be consistent with the UC and UP data types.

Some database-specific data types are supported that are not shown in the above table.

If DATATYPE information is not specified, the native data type is synthesized based on the column of the SQL data type or from the database system.

The SQL data type information in the USE AS clause identifies the value in the SIGNED column in the following instances:

- One character code is supplied
- No DATATYPE information is specified
- Native data type information is not obtained from the database system

USE AS

Identifies the SQL data type for the column.

The following table describes the data types for columns. The non-null SQLTYPE identifies the data type in internal control blocks and diagnostic trace information.

Table 7. SQL data type descriptions

Keyword identifier	Description	Maximum length	SQLTYPE
CHAR(length)	Fixed-length character string that contains mixed mode data.	255	452
VARCHAR(length)	Variable length character string that contains mixed mode data. A half-word length component precedes the character string and identifies the actual length of the data. The length field does not include the length of the length field.	32704	448
LONG VARCHAR	Long character string that contains mixed mode data. A half-word length component precedes the character string and identifies the actual length of the data. The length field does not include the length of the length field.	32704	456
GRAPHIC(length)	Fixed-length graphic string that is assumed to contain pure DBCS data without shift codes. The length is expressed in DBCS characters, and not bytes.	127	468

Table 7. SQL data type descriptions (continued)

Keyword identifier	Description	Maximum length	SQLTYPE
VARGRAPHIC(length)	Varying-length graphic string that is assumed to contain pure DBCS data without shift codes. A half-word length component precedes the graphic string and identifies the actual length of the data. The length is expressed in DBCS characters, and not bytes.	16352	464
LONG VARGRAPHIC	Long graphic string that is assumed to contain pure DBCS data without shift codes. A half-word length component precedes the graphic string and identifies the actual length of the data. The length is expressed in DBCS characters, and not bytes.	16352	472
INTEGER	Large integer.	Exactly 4	496
SMALLINT	Small integer.	Exactly 2	500
DECIMAL (precision,scale) or DECIMAL(precision)	Packed decimal data. A valid precision value is between 1 and 31. A valid scale value is between zero and the precision value.	31	484

Table 7. SQL data type descriptions (continued)

Keyword identifier	Description	Maximum length	SQLTYPE
FLOAT(precision)	Floating point number. Depending upon the precision, a column with a FLOAT data type takes on the attributes of a REAL or DOUBLE PRECISION data type. When precision is in the range of 1 to 21, the column is treated as a REAL. For precisions between 22 and 53, the column takes on DOUBLE PRECISION attributes. A precision of zero or greater than 53 is nonvalid.	4 or 8	480
REAL	Floating point number with a range of magnitude of approximately 5.4E 79 to 7.2E+75.	4	480
DOUBLE PRECISION	Floating pointer with a range of magnitude of approximately 5.4E 79 to 7.2E+75.	8	480

When you use a USE AS LONG VARCHAR or USE AS LONG VARGRAPHIC clause, you do not specify a length for the column. The length is based on the physical attributes that are obtained about the object record or segment where the column is. When you use a USE AS VARCHAR clause with a length greater than 254, that column is changed into a LONG VARCHAR column. The length in the clause is discarded. The maximum physical length is computed based on physical attributes. The same behavior occurs for a USE AS VARGRAPHIC clause when the length is greater than 127.

USE RECORD LENGTH

The USE RECORD LENGTH clause is specified after the USE AS clause for a VARCHAR, LONG VARCHAR, VARGRAPHIC or LONG VARGRAPHIC data type specification.

Indicates the varying string or graphic column contains the entire contents of the record that the column is associated with. The entire record content contains character data. The actual data contents were not important to any client application that accessed one of these columns that ran on an MVS system where no code page conversion was performed.

WITHOUT CONVERSION

Specifies that a field procedure does not exist for the column.

WITH CONVERSION

Identifies the name of the load module for a field procedure, if a field procedure is required. The field procedure name is a short identifier.

NULL IS *null_value*

Specifies a value that identifies when the contents of the column is null. By default, the data in a column never contains a null value. This situation is true even for variable length character or graphic columns where the length component indicates that there is no data in the variable length column.

A null value is a character string that can specify up to 16 characters of data. Specifies the value in hexadecimal format. The total length of the string is 35 characters.

A column contains null values based on *null-value*. If the start of the column data matches the null value, then that instance of the column is null. For example, if a column is defined as CHAR(10) and a null value of x'4040' is specified, the null value length is 2. If the first 2 bytes of the column contain spaces, the column is also null.

PRIMARY KEY

Identifies the column to be one of the columns that constitute the primary key for the table. This form of primary key identification is mutually exclusive with specifying a primary key at the end of the CREATE TABLE statement.

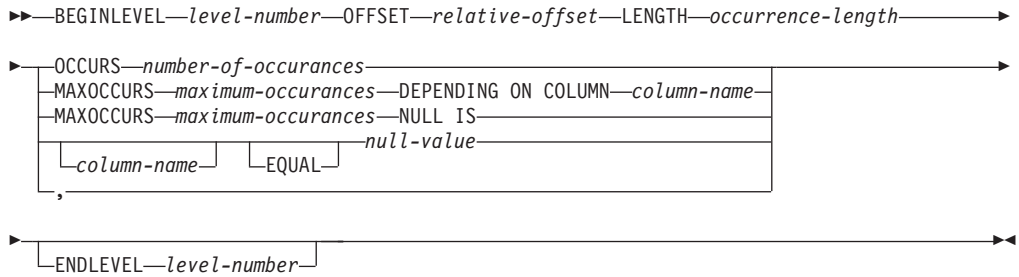
You can identify a primary key at the column level when the columns that make up the primary key for the table are defined in ordinal sequence within the table definition.

KSDS or files that are accessed through an alternate index of columns that map to the key offset and length information, which are determined during validation, are good candidate columns to identify as primary keys. VSAM ESDS or RRDS files that are not accessed through an alternate index are always poor candidates for primary keys.
You can use the PRIMARY KEY clause on the column definition to identify the primary key columns when the keys are defined in the same order.

Record arrays for VSAM

Record arrays identify and handle data that can repeat multiple times within a non-DB2 database or file system.

Syntax



Parameters

BEGINLEVEL *level-number*

Identifies the start of a record array.

The *level-number* on a BEGINLEVEL or ENDLEVEL definition identifies the nesting level of the record array within the parent fragment. Multiple record arrays can exist with the same *level-number*. However, these situations within a

single table represent a questionable mapping. Ideally you should create separate tables for each record array that exists at the same level. The reason is that when accessing these kinds of tables, for a single database access, the query processor can generate the Cartesian product for the maximum occurrences of all record arrays that are defined.

The following example shows a multiple record array. In this example, the EMPL-ADDRESS data item occurs three times and the EMPL-DEPENDENTS, EMPL-DEP-NAME and EMPL-DEP-DOB data items occur ten times. By default the query processor generates 30 rows for each EMPL-RECORD read.

```
01 EMPL-RECORD
   05 EMPL-SSN               PIC 9(9)
   05 EMPL_NAME
     10  EMPL-LAST           PIC X(30)
     10  EMPL_FIRST          PIC X(30)
     10  EMPL_MI             PIC X
   05 EMPL-ADDRESS           PIC X(30) OCCURS 3 TIMES
   05 EMPL-DEPENDENTS        OCCURS 10 TIMES
     10  EMPL-DEP-NAME       PIC X(30)
     10  EMPL-DEP-DOB        PIC 9(8)
```

Figure 8. Example of a multiple record array

If the information in the above example needs to be accessed in a single table definition, then two record array definitions are required. These record arrays are both level number 1 arrays because the employee address information exists first and repeats three times followed by the employee dependent information that occurs ten times.

OFFSET *relative-offset*
The relative offset identifies the relative starting position of the record array within its owning fragment definition. The owning fragment definition is either associated with the table or with a parent record array definition.

LENGTH *occurrence-length*

Iidentifies the length in bytes of one occurrence of the repeating data.

MAXOCCURS *maximum-occurrences* **DEPENDING ON COLUMN** *column-name*

Identifies a record array that occurs a variable number of times. The number of actual instances that exist in a particular database record is identified by the contents of a column definition that exists before the record array definition.

The following example uses an OCCURS DEPENDING ON COLUMN clause that defines a variably occurring record array. In this example, the EMPL-DEP-COUNT data item identifies the number of dependents that exist in the EMPL-DEPENDENTS record array.

```
01 EMPL-RECORD
   05 EMPL-SSN               PIC 9(9)
   05 EMPL_NAME
     10  EMPL-LAST           PIC X(30)
     10  EMPL_FIRST          PIC X(30)
     10  EMPL_MI             PIC X
   05 EMPL-ADDRESS           PIC X(30) OCCURS 3 TIMES
   05 EMPL-DEP-COUNT         PIC S9(4) COMP
   05 EMPL-DEPENDENTS        OCCURS 10 TIMES
                             DEPENDING ON EMPL-DEP-COUNT
     10  EMPL-DEP-NAME       PIC X(30)
     10  EMPL-DEP-DOB        PIC 9(8)
```

Figure 9. Example of an OCCURS DEPENDING ON defintion

You cannot map columns that occur after a variable length array, because the starting offsets of data items that are defined after an OCCURS DEPENDING ON definition are not at a fixed offset. The starting offsets float from one record instance to the next, based on the actual number of occurrences that are identified by the contents of the DEPENDING ON data item. The Classic federation servers do not currently support calculated field offsets, and you cannot access these fields.

The *maximum-occurrences* value must be a numeric value, and *column-name* must specify the name of a column in the table. The control column must contain a numeric value. The query process supports SQL types of SMALLINT, INTEGER, DECIMAL, or CHAR (if the column contains zoned decimal data).

MAXOCCURS *maximum-occurrences* **NULL IS**

Defines a record array that repeats a fixed number of times. You specify a value that denotes a null value. When a column in the record array contains the specified value, the content of the column is interpreted as null.

The *maximum-occurrences* value must be numeric. The *null-value* is a character string of up to 32 characters. The *null-value* can also be a 67-character hexadecimal string that consists of 64 hexadecimal nibbles, the leading x, and the surrounding quotation marks.

The minimal definition for this kind of array structure has the NULL IS keyword followed by the *null-value* string. With this syntax, a record array element contains null values based on the length of the *null-value* string. If the start of the column data matches the *null-value* then that instance of the repeating group of columns or record arrays is null. For example, a null-value of x'4040' is specified; the null value length is 2 and if the first two bytes of a record entry match the null value the entry is null. Processing of the array contents is skipped and the next element in the record array is inspected.

Elements or the array are processed like this:

1. The first element in the record array is located based on the relative-offset information.

   The physical location is computed based on the parent fragment offset information.
2. The second through *nth* instances starting locations are determined by incrementing the physical location by the occurrence length value.

Alternatively, you can check the entire contents of a record array to determine if it contains a null value by using the ALL keyword. For example, the syntax is NULL IS ALL null-value. With this form, only a single character of null value data can be supplied in either character or hexadecimal format.

When you add *column-name* EQUAL to the NULL IS clause, the null checking for a record array element is confined to the contents of a single column. Ideally, the column needs to exist within the record array. However, that column is not a requirement.

When column-level null checking is requested, the same kinds of options are available as with entry-level checking. The same kinds of errors can be generated. If the ALL keyword is not specified, then the contents of the column in each record array entry is checked to see if it matches the supplied value. If so, the entry is null and its contents are ignored. Likewise if the ALL keyword is supplied, only a single character null value can be specified. The entire contents of the column are checked to see if a record array instance is null and therefore ignored.

ENDLEVEL *level-number*
Identifies the end of the definition.

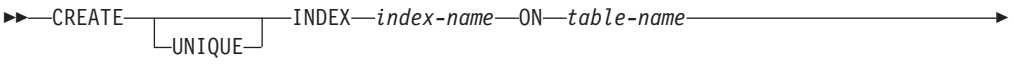
CREATE INDEX statement for VSAM

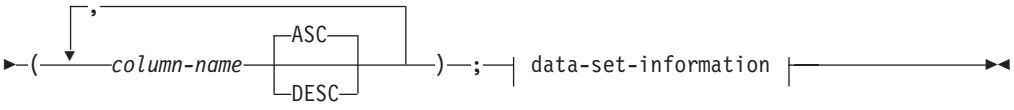
You can use the CREATE INDEX statement to define an index that references the columns that make up a VSAM KSDS primary key or the columns that map to an alternate index PATH definition.

Indexes identify columns in a table that correspond to a physical index that is in the source database. The query processor uses the contents of the columns that are referenced in a WHERE clause to create the index. The query processor attempts to build either a full key value to use in database access or a partial key value. The partial key can be used to perform a range scan against the target database to reduce the number of records that are accessed.

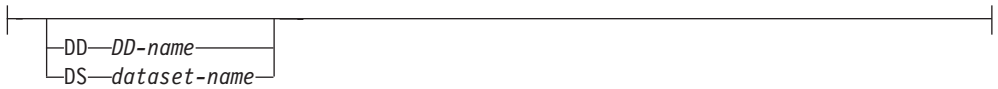
Although you might not be able to define an index against columns that correspond to the primary key, you can identify these columns as primary key columns. This primary key information does not affect existing connector index selection and access optimization. The primary key information is made available for use by front-end tools and for replication purposes, where key information is either required or beneficial.

Syntax





data-set-information::



Parameters

CREATE INDEX *index-name*

Identifies the SQL statement as an index definition statement. A unique index does not have any restrictions about the columns that make up the index. For example, a unique index column can contain null values.

If qualified, the index name is a two-part name, and the authorization ID that qualifies the name is the owner of the index. If an unqualified table name is supplied, the owner name is the authorization ID from the CURRENT SQLID special register.

ON *table-name*

Identifies the table for which the index is being defined. The table name can be a qualified or unqualified table name. For unqualified names, the table owner is from the CURRENT SQLID special register.

The table name is validated with the standard syntax checks that are associated with identifiers, and then the existence of the table is verified. Do not define an index on a view.

column-name **ASC/DESC**

Specifies column names that make up the index key. Optionally, you can identify whether the column is stored or accessed in ascending (ASC) or descending (DESC) key sequence. By default, the key is in ascending key sequence.

There are no fixed limits on the number of columns that you can identify as key columns for an index. The only requirements are as follows:

- The column must exist in the table.
- The column must map to what constitutes a key in the target database. In most cases, this determination is based on the starting offset and length of the column as compared to the starting offset and length of the key in the target database.
- The column cannot overlap another column within the key definition. This determination is based on the starting offset and length of a column as compared to the starting offsets and length of the other columns that are identified as key columns for the index definition.
- For the key column, generally varying length and graphic data types are not supported.

Key column verification is strict and enforces all rules for column names. Rule 4 enforcement (nonvalid data types) is enforced. Identification of any kind of varying-length character or graphic string is not allowed.

Data-set-information

Identifies the data set that accesses the columns that make up the table that *table_name* references. One common reason to create an index is when a VSAM KSDS file has an alternate index that is associated with it. The most likely scenario is that the table references the base cluster name, and some queries do not contain references in the WHERE clause to the VSAM primary key. But, the queries do contain columns in the WHERE clause that map to the key of the alternate index. For these queries, use the alternate index.

To define an alternate index, use either the DD clause or DS clause to identify the name of the alternate index path data set. Also, identify as the key columns those columns that map to the alternate index key. When a query that contains a WHERE clause references these key columns, automatic index selection

determines that the index is the best candidate for processing the query. Thus, the alternate index path accesses the actual VSAM data.

You can also define indexes to publish information to tools. For example, if you have a single column that maps the entire social security number and the three-column mapping for the components, you can define two index definitions. In the first index definition, identify the single column as the key column. In the second index definition, identify the three-component columns as the keys. In this example, identification of DD or DS information is not allowed. The DD or DS clauses only identify an alternate index path data set.

DD *DD-name*

Specifies the VSAM file that the table maps to. Specifies either a local file reference or a reference to a CICS file definition table (FDT) entry name.

If *DD-name* represents a local file reference, a DD clause that references the VSAM file must exist in the Classic federation data server JCL.

If *DD-name* references an FDT entry name, you must specify CICS connection information with the CICS_connection_information clause.

DD-name must correspond to a cluster or alternate index path definition for a VSAM ESDS, KSDS or RRDS data set.

DD-name is a short native identifier that conforms to the naming conventions for a z/OS JCL DD statement.

DS *data-set-name*

Specifies the VSAM file that the table definition accesses. Designates a VSAM cluster definition or a PATH component when the VSAM file is accessed from a VSAM alternate index.

The data set name must correspond to a cluster or alternate index path definition for a VSAM ESDS, KSDS or RRDS data set.

The name can be 1 to 44 characters in length and follows z/OS naming conventions for VSAM data sets.

Example

The following is an example of a CREATE INDEX statement for VSAM.

```
CREATE UNIQUE INDEX "DBA"."EMPLOYEE_EP_IDX1" ON "DBA"."EMPLOYEE_EP" ("EMPL_SSN" ASC);
```

ALTER TABLE statement for VSAM

You can use the ALTER TABLE statement to alter the definition of a table.

Authorization

You must have the one of the following authorities to run the ALTER TABLE statement:

- SYSADM
- DBADM for the database type that is referenced in the DBNAME column for the table being altered
- Ownership of the table being altered

Syntax

Parameters

table-name
Identifies the table for which the DATA CAPTURE flag is being altered. The table name can be a qualified or unqualified table name. If an unqualified name is supplied, the table owner is from the CURRENT SQLID special register.

DATA CAPTURE

Indicates that the ALTER TABLE statement is setting the value of the DATA CAPTURE flag.

CHANGES
Allows a change-capture agent to capture changes to the non-relational data that the table is mapped to.

The following restrictions apply:

- Change capture is not allowed if the table references an alternate index.
- Change capture with an NVA change-capture agent is allowed only if the table is defined with a data set name, rather than with a DD name.

NONE
Disallows change-capture agents to capture changes to the non-relational data that the table is mapped to.

Example

The following is an example of an ALTER TABLE statement for VSAM.

```
ALTER TABLE "DBA"."EMPLOYEE_EP" DATA CAPTURE CHANGES;
```

Chapter 4. SQL security

Security of your data is particularly important in an SQL-based DBMS, because interactive SQL makes database access very easy. The security requirements of production databases can include data in any given table that is accessible to some users, but denied to others, and allow some users to update data in a table, while others can only view data.

Overview of SQL security

SQL security in federation, event publishing, and replication is similar to security in DB2 databases.

Implementing security and enforcing security restrictions are the responsibility of the DBMS software. SQL defines an overall framework for database security, and SQL statements specify security access and restrictions.

SQL security involves the following key concepts:

- Users are the actors in the database. When the DBMS retrieves, inserts, deletes, or updates data, it does so on behalf of a user or group of users. The DBMS permits or denies user actions depending on which user makes the request. You can define users and user groups based on categories of administrative authorities.
- Database objects, such as tables, views, and stored procedures are the objects to which SQL security can be applied.
- Privileges are the actions that a user is permitted to perform against a particular database object. For example, a user might have permission to select and insert rows in one table, but lack permission to delete or update rows in that table. These privileges are allowed or prohibited by using the GRANT and REVOKE SQL statements.
- SQL security and the SAF exit work together to ensure that the user ID and its password are checked before allowing access to particular database objects.
- SQL security is required for federation, event publishing, and replication.

Related concepts

Security: SAF exit
Use the SAF exit to verify that a user has authority to access a physical file or PSB referenced in an SQL query. The SAF exit also verifies that a user has authority to execute a stored procedure program.

Authorization

In federation, event publishing, and replication, each user ID is associated with a particular authority level.

You can assign users to the following authority levels:

SYSADM
The system administrator has privileges for all objects and has the ability to grant authority to other users. The first user to run the metadata utility is granted SYSADM authority.

SYSOPR
The system operator has remote-operator privileges to display and manage an active data server.

DISPLAY
This user or user group has remote-operator privileges for display commands on an active data server.

DBADM
The database administrator has mapping and view-creation privileges for specific database types.

PUBLIC
This user or user group is limited to privileges that are explicitly granted to its user name or PUBLIC.

You assign privileges to individual users or groups of users based on an authorization ID by using the SQL GRANT statement. The authorization ID determines whether the statement is permitted or prohibited by the DBMS. In production databases, the database administrator assigns authorization IDs.

Authorization requirements for SQL statements

To issue GRANT, REVOKE, SELECT, EXECUTE, INSERT, UPDATE, and DELETE statements, you must have specific authorization.

The following table describes authority levels that are required to issue each of these statements.

Table 8. Authorization requirements for SQL statements

Statement	Authority required
{GRANT | REVOKE}	To grant a privilege, you must have SYSADM authority, or you must be granted the privilege itself with the WITH GRANT option. To revoke a privilege, you must have SYSADM authority or be the user who originally granted the privilege being revoked. The BY ALL clause requires SYSADM authority because you are revoking revoke privileges that are granted by users other than yourself.
SELECT	SYSADM authority or the specific privilege is required. SELECT authority on all tables and views is referenced in the SELECT statement.
EXECUTE	SYSADM authority or the specific privilege is required. EXECUTE authority is on the procedure.
INSERT	SYSADM authority or the specific privilege is required. INSERT authority is on the table.
UPDATE	SYSADM authority or the specific privilege is required. UPDATE authority is on the table and SELECT authority on all tables is referenced in the WHERE clause.
DELETE	SYSADM authority or the specific privilege is required. DELETE authority is on the table. SELECT authority is on all tables that are referenced in the WHERE clause.

Related reference

Verifying SMF exit output
DSECT fields map the output data from the SMF exit accounting routine. Understanding the field definitions enables you to verify this output data.

Database objects in SQL security

Catalog database types are database objects to which security can be applied.

To manage or secure the metadata utility for mapping purposes, the following implicit database names are associated with metadata catalogs:

$ADABAS
For Adabas database mappings

$CFI For system catalog

$DATACOM
For CA-Datacom database mappings

$IDMS
For CA-IDMS database mappings

$IMS For IMS database mappings

$SEQUENT
For sequential database mappings

$SP For stored procedure definitions

$VSAM
For VSAM database mappings

These metadata catalogs map one-to-one with the connectors, with the exception of $CFI.

You can specify database types in the GRANT DBADM statement. For example:

```
GRANT DBADM ON DATABASE $IMS TO USER1
```

To map tables, you must have the SYSADM authority to run the metadata utility, because the metadata utility does not have security-access checking at the database level. Grant DBADM authority only for DROP commands.

Related concepts

Metadata catalog
The information that you generate from the Classic Data Architect is stored in metadata catalogs.

Defining user privileges

Privileges are the set of actions that a user can perform. The SQL GRANT and REVOKE statements assign privileges.

You can use the Classic Data Architect to grant and revoke one or more of the following privileges:

- System
- Database

- Stored procedures
- Tables and views

The GRANT and REVOKE are executable statements that can be dynamically prepared.

A specific grantor can grant or revoke specific privileges to specific users, with the restriction that a privilege can only be revoked if it has first been granted.

Related concepts

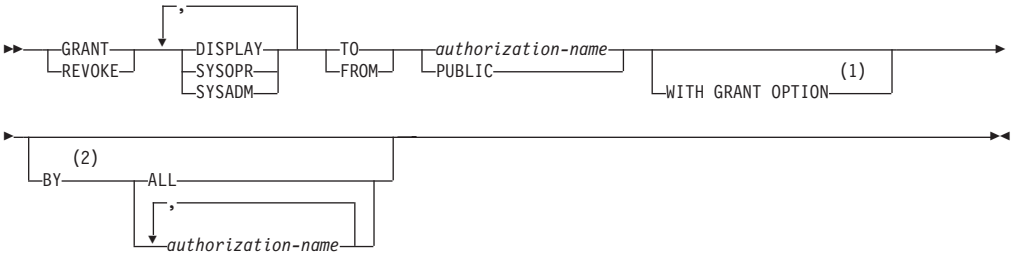
Stored procedures
A stored procedure is an application program that performs work that SQL SELECT, INSERT, UPDATE, and DELETE operations cannot perform. A client application invokes a stored procedure application by issuing an SQL CALL statement.

Security: SAF exit
Use the SAF exit to verify that a user has authority to access a physical file or PSB referenced in an SQL query. The SAF exit also verifies that a user has authority to execute a stored procedure program.

System privileges

System privileges allow or deny access to a specific set of catalogs within a data server.

Syntax:



Notes:

1 GRANT only

2 REVOKE only. Only revokes privileges granted by that user

The following table describes the statement parameters.

Table 9. Parameter descriptions for the GRANT and REVOKE statement.

Parameter	Description
{GRANT | REVOKE}	GRANT or REVOKE privileges to user IDs or groups of user IDs.
DISPLAY	GRANT or REVOKE the privileges to remotely issue all forms of the DISPLAY command to a data server.

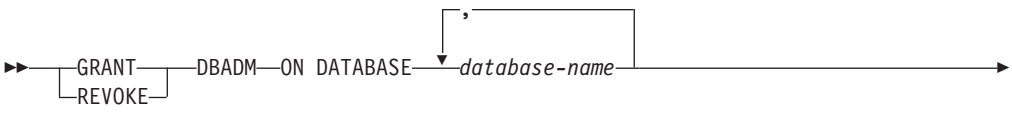
Table 9. Parameter descriptions for the GRANT and REVOKE statement. (continued)

Parameter	Description
SYSOPR	GRANT or REVOKE the privilege to remotely issue all commands to a data server including the commands to start and stop services and shut down the data server. Commands issued from the system console are not secured.
SYSADM	GRANT or REVOKE system administrator authority.
{TO | FROM} {*authorization-name* | PUBLIC}	GRANT or REVOKE authority to a particular user or group of users, or to all users or groups of users on a system.
WITH GRANT OPTION	GRANT authority to a particular user or group of users to GRANT authority to other users or other groups of users in the system. Although this option can be specified when granting the SYSADM privilege, it has no effect on SYSADM privileges because SYSADM has ALL access privileges available in the data server.
BY ALL *authorization-name*	BY revokes each named privilege that was explicitly granted to some named user or group of users by one of the named grantors. Only an authorization ID with SYSADM authority can use BY, even if the authorization ID names only itself in the BY clause. ALL then revokes each named privilege from all named users or group of users. *authorization-name* lists one or more authorization IDs of users or group of users who were the grantors of the privileges named. Do not use the same authorization ID more than once. Each grantor listed must have explicitly granted some named privilege to all named users or group of users.

Database privileges

Authorization IDs with DBADM privileges can grant and revoke specific privileges within a particular database.

Syntax:



Notes:

1 GRANT only

The following table describes the statement parameters.

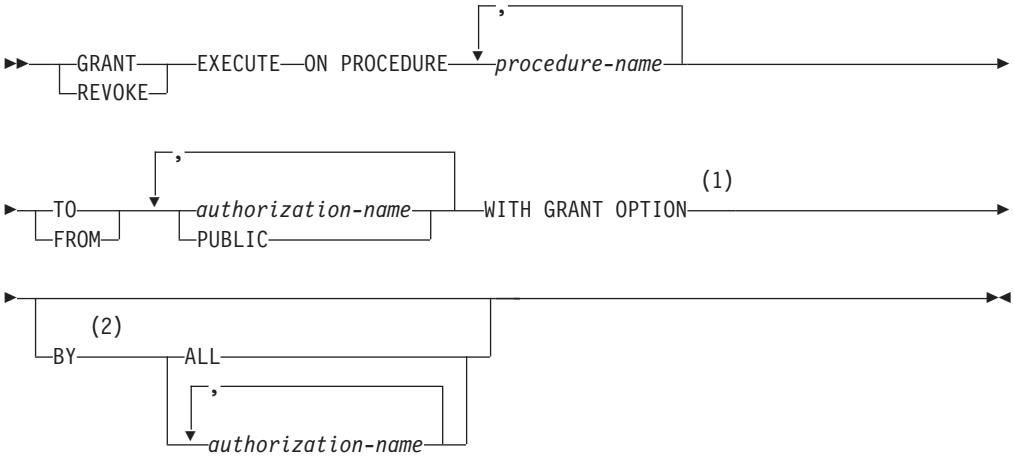
Table 10. Parameter descriptions for the GRANT and REVOKE statement.

Parameter	Description
GRANT ON DATABASE *database-name*	Identifies database types on which privileges are to be granted. For each named database type the grantor must have all the specified privileges with the GRANT option. This privilege secures the mappings of tables and dropping of mapped tables. The types are as follows: **$ADABAS** For Adabas mappings **$CFI** For metadata catalog mappings **$DATACOM** For CA-Datacom mappings **$IDMS** For CA-IDMS mappings **$IMS** For IMS mappings **$SEQUENT** For sequential **$SP** for stored procedures **$VSAM** For VSAM
REVOKE ON DATABASE *database-name*	Identifies the database type on which you revoke the privileges. For each database type, you or the indicated grantors must have granted at least one of the specified privileges on that database to all identified users (including PUBLIC, if specified). The same database type must not be identified more than once. The database-names are the same as those listed for the GRANT ON DATABASE database-name option.
{TO | FROM} {*authorization-name* | PUBLIC}	Specifies to what authorization IDs the privileges are granted or revoked. The *authorization-name* variable lists one or more authorization IDs.
WITH GRANT OPTION	Allows the named users to grant the database privileges to others. The user can specifically grant any privilege belonging to that authority. If you omit WITH GRANT OPTION, the named users cannot grant the privileges to others unless they have that authority from some other source.

Stored procedure privileges

Stored procedure privileges allow or deny access to a particular stored procedure.

Syntax:



Notes:

1 GRANT only

2 REVOKE only. Only revokes privileges granted by that user

The following table describes the statement parameters.

Table 11. Parameter descriptions for the GRANT and REVOKE statement.

Parameter	Description
{GRANT | REVOKE}	Grants or revokes authority to run a stored procedure.
ON PROCEDURE *procedure-name*	Identifies the procedure for which you grant or revoke privileges.
{TO | FROM} {*authorization-name* | PUBLIC}	Specifies to or from which authorization IDs the privileges are granted or revoked. The *authorization-name* variable lists one or more authorization IDs.
WITH GRANT OPTION	Allows the named users to grant the stored procedure privileges to others. The user can specifically grant any privilege belonging to that authority. If you omit WITH GRANT OPTION, the named users cannot grant the privileges to others unless they have that authority from some other source.

Table 11. Parameter descriptions for the GRANT and REVOKE statement. (continued)

Parameter	Description
BY ALL *authorization-name*	BY revokes each named privilege that was explicitly granted to some named user or group of users by one of the named grantors. Only an authorization ID with SYSADM authority can use BY, even if the authorization ID names only itself in the BY clause. ALL then revokes each named privilege from all named users or group of users. *authorization-name* lists one or more authorization IDs of users or group of users who were the grantors of the privileges named. Do not use the same authorization ID more than once. Each grantor listed must have explicitly granted some named privilege to all named users or group of users.

Table and view privileges

Table and view privileges allow or deny access to specific tables and views.

GRANT syntax for tables and views

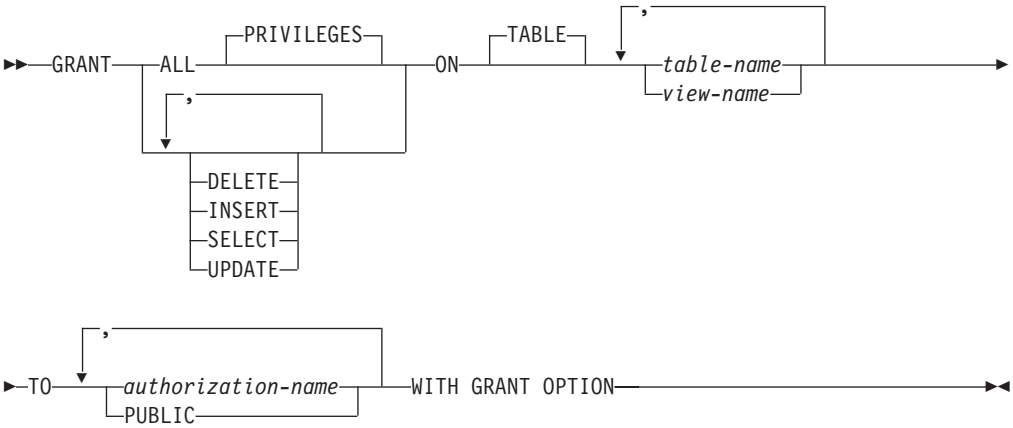


Table 12. Parameter descriptions for the GRANT statement.

Parameter	Description
GRANT {ALL | ...} PRIVILEGES	Grants all table or view privileges for which you have GRANT authority, for the tables and views named in the ON clause.
DELETE	Grants privileges to use the DELETE statement.
INSERT	Grants privileges to use the INSERT statement.
SELECT	Grants privileges to use the SELECT statement.

Table 12. Parameter descriptions for the GRANT statement. (continued)

Parameter	Description
UPDATE	Grants privileges to use the UPDATE statement.
ON TABLE {*table-name* | *view-name*}	Names the tables or views on which you are granting the privileges. The list can be a list of table names or view names, or a combination of the two.
{TO | FROM} {*authorization-name* | PUBLIC}	Specifies to or from which authorization IDs the privileges are granted or revoked. *authorization-name* lists one or more authorization IDs.
WITH GRANT OPTION	Allows the named users to grant the table/view privileges to others. Granting an administrative authority with this option allows the user to specifically grant any privilege belonging to that authority. If you omit WITH GRANT OPTION, the named users cannot grant the privileges to others unless they have that authority from some other source.

REVOKE syntax for tables and views

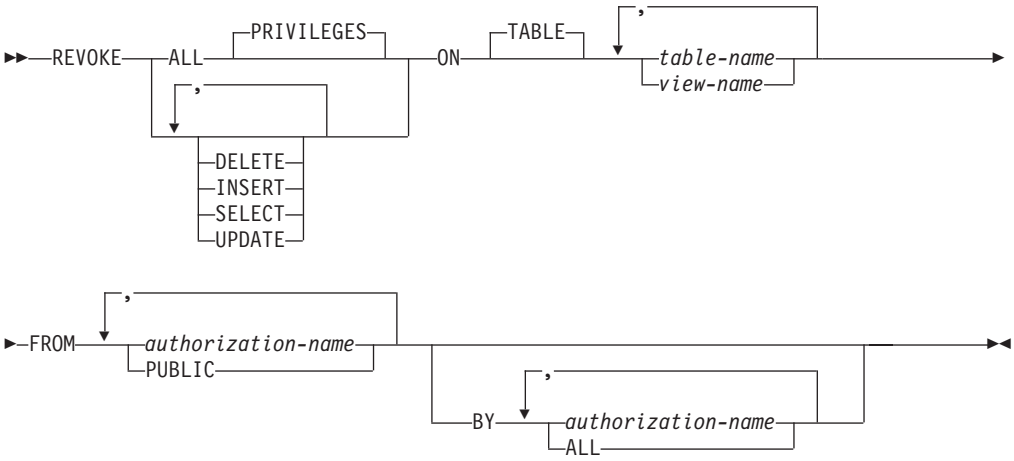


Table 13. Parameter descriptions for the REVOKE statement.

Statement	Description
REVOKE {ALL | ...} PRIVILEGES	Revokes all table or view privileges for which you have GRANT authority, for the tables and views named in the ON clause.
DELETE	Revokes privileges to use the DELETE statement.
INSERT	Revokes privileges to use the INSERT statement.
SELECT	Revokes privileges to use the SELECT statement.

Table 13. Parameter descriptions for the REVOKE statement. (continued)

Statement	Description
UPDATE	Revokes privileges to use the UPDATE statement.
ON TABLE {table-name | view-name}	Names the tables or views on which you are granting the privileges. The list can be a list of table names or view names, or a combination of the two.
FROM {*authorization-name* | PUBLIC}	Specifies to what authorization IDs the privileges are revoked. *authorization-name* lists one or more authorization IDs.
BY {ALL | *authorization-name*}	BY revokes each named privilege that was explicitly granted to some named user or group of users by one of the named grantors. Only an authorization ID with SYSADM authority can use BY, even if the authorization ID names only itself in the BY clause. ALL then revokes each named privilege from all named users or group of users. *authorization-name* lists one or more authorization IDs of users or group of users who were the grantors of the privileges named. Do not use the same authorization ID more than once. Each grantor listed must have explicitly granted some named privilege to all named users or group of users.

SAF and SMF system exits for SQL security

In addition to authorizations and privileges for SQL security, you need to use the System Authorization Facility (SAF) security exit. The SAF exit is required to validate passwords for data server connections.

With the SAF exit, you can administer SQL security at a group level. The group name, in addition to the user ID, determines whether users are authorized to perform the operation they attempt. A group name makes security administration easier because you only need to grant and revoke privileges to and from the group name for all users that are associated with that group name. When you need to authorize access to new users, you can use your external security package and assign the new user to the default group name.

In addition, the query processor calls the System Management Facility (SMF) exit when an authorization violation is detected. The SMF exit generates an SMF record that logs the user ID, type of authorization failure, and the name of the object for which authorization failed.

Related concepts

Security: SAF exit
Use the SAF exit to verify that a user has authority to access a physical file or PSB referenced in an SQL query. The SAF exit also verifies that a user has authority to execute a stored procedure program.

Accounting: SMF exit

The SMF exit is used to report wall-clock time and CPU time for an individual user session with a query processor task. Additionally, if SQL security is active and an authorization violation is detected by the query processor, the exit is called to log the violation.

Related reference

Verifying SMF exit output

DSECT fields map the output data from the SMF exit accounting routine. Understanding the field definitions enables you to verify this output data.

Chapter 5. Views

In Classic federation and change capture, views can provide alternative ways to work with the data.

Database tables define the structure and organization of the data it contains. Using SQL, you can look at the stored data in other ways by defining alternative views of the data. A *view* is an SQL query that is stored in the database and assigned a name, similar to a table name. The results of the stored query are then visible through the view. With SQL, you can access these query results as if the results were a real table in the database. You can also use views as sources for publications and subscriptions.

In Classic federation, views allow you to manage your data in these ways:

- Tailor the appearance of a database so that different users see it from different perspectives.
- Restrict access to data, allowing different users to see only certain rows or certain columns of a table.
- Simplify database access by presenting the structure of the stored data in the way that is most natural for each user.

In change capture, views allow you to manage your data in these ways:

- Tailor change capture from a database for different purposes.
- Capture changes from source data with multiple record layouts.

Related concepts

Classic Data Architect
The Classic Data Architect is the redesigned administrative tool introduced in Version 9 that replaces the Classic Data Mapper.

Record types and COBOL example

Many COBOL record layouts contain layout information to map different record types within a single physical file. Whenever record types are defined in a copybook, create a separate table for each value. The record type field can contain and map only the COBOL fields that are associated with a specified value.

As a simple example, a single VSAM file stores both employee and address records. The correct record interpretation is managed by comparing a record type field in the COBOL record layout to a set of known values. The following COBOL definition shows how this comparison is done by using a REDEFINES clause:

```
01 EMPLOYEE-ADDRESS-RECORD.
05 RECORD-TYPE       PIC X.
   88 RECORD-IS-EMPLOYEE   VALUE 'E'.
   88 RECORD-IS-ADDRESS    VALUE 'A'.
05 EMPLOYEE-INFORMATION.
   10 LAST-NAME      PIC X(20).
   10 FIRST-NAME     PIC X(20).
   10 DATE-OF-BIRTH  PIC 9(8).
   10 MONTHLY-SALARY PIC S9(5)V99 COMP-3.
   10 FILLER         PIC X(48).
05 ADDRESS-INFORMATION REDEFINES EMPLOYEE-INFORMATION.
   10 ADDRESS-LINE-1 PIC X(30).
```

```
   10 ADDRESS-LINE-2 PIC X(30).
   10 ADDRESS-CITY   PIC X(20).
   10 ADDRESS-STATE  PIC XX.
   10 ADDRESS-ZIP    PIC 9(5).
```

A COBOL REDEFINES clause redefines data in the identical record locations of a record layout. In this case, EMPLOYEE-INFORMATION and ADDRESS-INFORMATION both start at location 2 in the record. To accurately map the COBOL record above, two distinct SQL mappings are necessary for the file. The first mapping consists of columns for RECORD-TYPE, LAST-NAME, FIRST-NAME, DATE-OF-BIRTH, and MONTHLY-SALARY. The second mapping consists of RECORD-TYPE, ADDRESS-LINE-1, ADDRESS-LINE-2, ADDRESS-CITY, ADDRESS-STATE, and ADDRESS-ZIP. Each of these mappings is used to create a separate table definition in the metadata catalog.

By default, in response to an SQL query or data change, every record in the file matches the defined table. If the underlying data does not match the definition, errors in processing occur or nonvalid information is returned. In the case described above, an address record retrieved by using the employee table definition produces a data error because the MONTHLY-SALARY field contains character address data instead of numeric salary data.

To process record instances accurately with different layouts, you must apply record selection criteria to the underlying data. If the selection criteria do not match, records are skipped because they do not apply to the defined mapping.

Example: The following CREATE TABLE statements define the EMPLOYEE and ADDRESS layouts above:

```
CREATE TABLE VSAM.EMPLOYEE_NAME DBTYPE VSAM
 DS "VSAMEMP.KSDS"
(
 RECORD_TYPE SOURCE DEFINITION
  DATAMAP OFFSET 0 LENGTH 1
  DATATYPE C
  USE AS CHAR(1),
 LAST_NAME SOURCE DEFINITION
  DATAMAP OFFSET 1 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 FIRST_NAME SOURCE DEFINITION
  DATAMAP OFFSET 21 LENGTH 20
  DATATYPE C
  USE AS CHAR(20),
 DATE_OF_BIRTH SOURCE DEFINITION
  DATAMAP OFFSET 41 LENGTH 8
  DATATYPE UC
  USE AS CHAR(8),
 MONTHLY_SALARY SOURCE DEFINITION
  DATAMAP OFFSET 49 LENGTH 4
  DATATYPE P
  USE AS DECIMAL(7 , 2));

CREATE TABLE VSAM.EMPLOYEE_ADDRESS DBTYPE VSAM
 DS "VSAMEMP.KSDS"
(
 RECORD_TYPE SOURCE DEFINITION
  DATAMAP OFFSET 0 LENGTH 1
  DATATYPE C
  USE AS CHAR(1),
 ADDRESS_LINE_1 SOURCE DEFINITION
  DATAMAP OFFSET 1 LENGTH 30
  DATATYPE C
```

```
 USE AS CHAR(30),
ADDRESS_LINE_2 SOURCE DEFINITION
 DATAMAP OFFSET 31 LENGTH 30
 DATATYPE C
 USE AS CHAR(30),
ADDRESS_CITY SOURCE DEFINITION
 DATAMAP OFFSET 61 LENGTH 20
 DATATYPE C
 USE AS CHAR(20),
ADDRESS_STATE SOURCE DEFINITION
 DATAMAP OFFSET 81 LENGTH 2
 DATATYPE C
 USE AS CHAR(2),
ADDRESS_ZIP SOURCE DEFINITION
 DATAMAP OFFSET 83 LENGTH 5
 DATATYPE UC
 USE AS CHAR(5));
```

Given the example, there are two choices in Classic federation:

Create two tables, each table mapping a different record layout

SQL queries or updates that are issued by client applications need to contain a WHERE clause that selects only the proper record type for each table mapped.

For example, you can use a SELECT statement similar to the following statement:

```
SELECT * FROM VEMPL_NAME WHERE RECORD_TYPE='E';
```

Create two tables and create a view on each table

The following view creation statements apply selection logic for the table mappings:

```
CREATE VIEW VEMPL_NAME AS
 SELECT * FROM EMPLOYEE_NAME WHERE RECORD_TYPE = 'E';
CREATE VIEW VEMPL_ADDRESS AS
 SELECT * FROM EMPLOYEE_ADDRESS WHERE RECORD_TYPE = 'A';
```

You can then use a SELECT statement like this:

```
SELECT * FROM VEMPL_NAME;
```

Given the example, there are two choices for change capture:

Create two tables and create a view on each table

The following view creation statements apply selection logic for the table mappings:

```
CREATE VIEW VEMPL_NAME AS
 SELECT * FROM EMPLOYEE_NAME WHERE RECORD_TYPE = 'E';
CREATE VIEW VEMPL_ADDRESS AS
 SELECT * FROM EMPLOYEE_ADDRESS WHERE RECORD_TYPE = 'A';
```

You have to then alter the views for change capture. When you create the tables and views in the Classic Data Architect, the ALTER VIEW statements are generated when you generate the DDL for the tables and views.

Create two tables and use the record selection exit CACRCSEL

For information about using CACRCSEL, see Record selection exit for multiple record layouts.

Views and the query processor in Classic federation

When the query processor encounters a reference to a view in an SQL statement, it finds the definition of the view in the database.

Then the query processor translates the request that references the view into an equivalent request against the source tables of the view and carries out that request. The query processor maintains the illusion of the view while the query processor maintains the integrity of the source tables.

Advantages and disadvantages of views in Classic federation

Views provide a variety of benefits and can be useful for many types of databases.

In a personal computer database, views are usually a convenience, defined to simplify requests to databases. In a production database installation, views can play an important role in defining the structure of the database for users or groups of users and can enforce the database security.

Views provide the following benefits:

- Built-in security: Gives each user permission to access the database only through a small set of views that contain the specific data the user or group of users is authorized to see, restricting user access to other data.
- Simplicity for queries: A view can draw data from several tables and present a single table, simplifying the information and turning multi-table queries into single-table queries for a view.
- Simplicity in structure: Views give users a specific view of the database structure, presenting the database as a set of virtual tables specific to particular users or groups of users.
- Stabilization of information: Views present a consistent, unchanged image of the database structure, even if underlying source tables are changed.

Although there are many advantages to views, the main disadvantage to using views rather than real tables is performance degradation. Because views only create the appearance of a table, not a real table, the query processor must translate queries against the view into queries against the underlying source tables. If the view is defined by a complex, multi-table query, even simple queries against the view become complicated joins that can take a long time to complete.

Joined views in Classic federation

One of the most frequent reasons for using views is to simplify multi-table queries.

Restriction: For change capture, you cannot use views that reference more than one table.

By specifying a two-table or a three-table query in the view definition, you can create a joined view. Joined views draw their data from two or three different tables and present the query results as a single virtual table. After you define the view, you can use a simple, single-table query against the view for requests that otherwise require a two-or-more-table join.

Example

A user often runs queries against a particular table, such as the ORDERS table. The user does not want to work with employee numbers, but wants a view of the ORDERS table that has names instead of numbers. You can create the following view:

```
CREATE VIEW ORDER_INFO (ORDER_NUM, COMPANY, REP_NAME, AMOUNT) AS
    SELECT ORDER_NUM, COMPANY, NAME, AMOUNT
     FROM ORDERS, CUSTOMERS, SALESREPS
    WHERE CUST = CUST_NUM
     AND REP = EMPL_NUM
```

This view is defined by a three-table join. As with a grouped view, processing required to create this virtual table is substantial. Each row of the view is created from a combination of one row from the ORDERS table, one row from the CUSTOMERS table, and one row from the SALESREPS table.

Although this view has a complex definition, it can be very valuable. For example, you can create the following query against this view:

```
SELECT REP_NAME, COMPANY SUM(AMOUNT)
  FROM ORDER_INFO
 GROUP BY REP_NAME, COMPANY
```

That query generates a report of orders that are grouped by salesperson:

```
REP_NAME   COMPANY     SUM(AMOUNT)
-----------------------------------------------
Bill Adams ACME Mfg.   $35,582.00
Bob Burns  JCP Inc.    $24,343.00
Dan Jones  First Corp. $75,000.00
```

This query is now a single-table SELECT statement, which is far simpler than the original three-table query. Also, the view makes it easier to see the operations in the query. The query processor, however, still must work harder to generate the query results for this single-table query against the view as it would to generate query results for the same three-table query. However, for the actual user, it is much easier to write and understand a single-table query that references the view.

CREATE VIEW statement

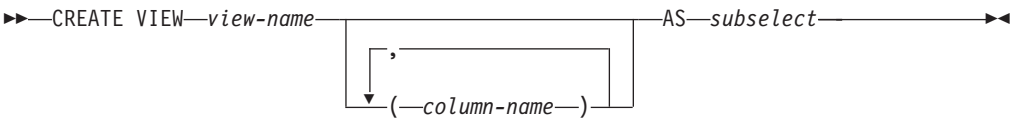
You can create and manage DB2 views by using the Classic Data Architect, the metadata utility, or any client that is connected to the data server.

Although most clients can create a view with standard SQL processing, the Classic Data Architect is a more controlled mechanism for creating and managing views. For this reason, use the Classic Data Architect to create views.

The data type, length, and other characteristics of the columns are derived from the definition of the columns in the source tables.

The CREATE VIEW statement can be embedded in an application program or issued interactively. The statement is an executable statement that can be dynamically prepared.

Syntax diagram



Parameters

view-name
Assigns a name to the view. The name cannot identify a table, view, alias, or synonym that exists on the current server. The name can be a two-part name. The authorization name that qualifies the name is the owner of the view.

column-name
Names the columns in the view. If you specify a list of column names, the list must consist of as many names as there are columns in the result table of the subselect. Each name must be unique and unqualified. If you do not specify a list of column names, the columns of the view inherit the names of the columns of the result table of the subselect.

You must specify a list of column names if the result table of the subselect has duplicate column names or an unnamed column (a column that is derived from a constant, function, or expression that is not given a name by the AS clause).

AS*subselect*
Defines the view. At any time, the view consists of the rows that result if the subselect is run. The subselect cannot refer to host variables or include parameter markers (question marks).

A query that contains either a UNION or an ORDER BY clause is not a valid subselect.

Related concepts

Classic Data Architect
The Classic Data Architect is the redesigned administrative tool introduced in Version 9 that replaces the Classic Data Mapper.

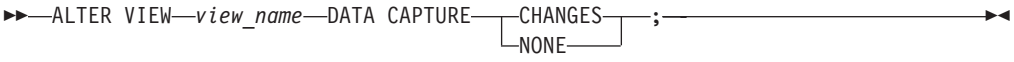
ALTER VIEW statement

To use a view as a source for a publication in Classic event publishing or for a subscription in Classic replication, you must alter the view to support change capture.

Certain views might not be valid for change capture for your particular data source.

Although most clients can create a view with standard SQL processing, the Classic Data Architect is a more controlled mechanism for creating and managing views. For this reason, use the Classic Data Architect to alter views.

Syntax diagram



Parameters

view_name
Identifies the view for which the DATACAPTURE parameter is being altered. The *view-name* can be a qualified or unqualified view name. If an unqualified name is supplied, the view owner is obtained from the CURRENT SQLID special register.

DATA CAPTURE

CHANGES
Turns data capture on for the view and stores a value of Y in the DATACAPTURE column in the SYSIBM.SYSTABLES row for the view.

NONE
Turns data capture off and stores a space in the DATACAPTURE column.

It is not an error to set the DATACAPTURE column to its existing value. Additionally, regardless of referenced table type, the DATACAPTURE flag can also be turned off even if the view definition is not eligible for data capture.

At least one of the following permissions is required to run the ALTER VIEW statement:

- SYSADM
- DBADM for the database type of the table that is referenced in the view
- Ownership of the view being altered

The following restrictions apply to change capture for a view:

- The view cannot reference more than one table. This restriction includes tables in the FROM clause or the WHERE clause as in the case of sub selects.
- The view cannot reference another view.
- The select list for the view must be ″SELECT *″.

Related concepts

Classic Data Architect
The Classic Data Architect is the redesigned administrative tool introduced in Version 9 that replaces the Classic Data Mapper.

DROP VIEW statement

To drop a view, you must use the DROP VIEW statement.

This statement provides detailed control over what happens when a user attempts to drop a view when the definition of another view depends on it.

Example: Two views on the SALESREPS table are created by these CREATE VIEW statements:

```
CREATE VIEW EASTREPS AS
   SELECT *
    FROM SALESREPS
   WHERE REP_OFFICE IN  (11, 12, 13)
CREATE VIEW NYREPS AS
   SELECT *
    FROM EASTREPS
   WHERE REP_OFFICE = 11
```

The following DROP VIEW statement removes both views as well as any views that depend on their definition from the database:

```
DROP VIEW EASTREPS
```

The CASCADE and RESTRICT parameters are not directly supported in the DROP VIEW syntax. However, the DROP VIEW deletes dependent views along with those specified in the DROP VIEW.

Related concepts


Classic Data Architect
The Classic Data Architect is the redesigned administrative tool introduced in Version 9 that replaces the Classic Data Mapper.

Accessing information about IBM

IBM® has several methods for you to learn about products and services.

You can find the latest information on the Web at www.ibm.com/software/data/sw-bycategory/subcategory/SWB50.html

- Product documentation in PDF and online information centers
- Product downloads and fix packs
- Release notes and other support documentation
- Web resources, such as white papers and IBM Redbooks™
- Newsgroups and user groups
- Book orders

To access product documentation, go to this site:

publib.boulder.ibm.com/infocenter/iisclzos/v9r1/

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order.
- To order publications by telephone in the United States, call 1-800-879-2755.

To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide.

Providing comments on the documentation

Please send any comments that you have about this information or other documentation.

Your feedback helps IBM to provide quality information. You can use any of the following methods to provide comments:

- Send your comments using the online readers' comment form at www.ibm.com/software/awdtools/rcf/.
- Send your comments by e-mail to comments@us.ibm.com. Include the name of the product, the version number of the product, and the name and part number of the information (if applicable). If you are commenting on specific text, please include the location of the text (for example, a title, a table number, or a page number).

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM trademarks and certain non-IBM trademarks are marked at their first occurrence in this document.

See http://www.ibm.com/legal/copytrade.shtml for information about IBM trademarks.

The following terms are trademarks or registered trademarks of other companies:

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft®, Windows®, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel®, Intel Inside® (logos), MMX and Pentium® are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Linux® is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product or service names might be trademarks or service marks of others.

Index

T

U

V

Printed in USA

SC19-1128-00